

COMBINED DECLARATION AND POWER OF ATTORNEY

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name,

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled SYNCHRONIZATION OF RECURRING RECORDS IN INCOMPATIBLE DATABASES, the specification of which

☒ is attached hereto.

☐ was filed on _____ as Application Serial No. _____
and was amended on _____.

☐ was described and claimed in PCT International Application No. _____
filed on _____ and as amended under PCT Article 19 on _____.

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose all information I know to be material to patentability in accordance with Title 37, Code of Federal Regulations, §1.56(a).

I hereby appoint the following attorneys and/or agents to prosecute this application and to transact all business in the Patent and Trademark Office connected therewith: G. Roger Lee, Reg. No. 28,963.

Address all telephone calls to G. Roger Lee, Esq. at telephone number 617/542-5070.

Address all correspondence to G. Roger Lee, Esq., Fish & Richardson P.C., 225 Franklin Street , Boston, MA 02110-2804.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patents issued thereon.

Full Name of Inventor: David J. Boothby

Inventor's Signature: _____ Date: _____

Residence Address: 12 Thoreau Drive, Nashua, NH 03062

Citizen of: United States

Post Office Address: 12 Thoreau Drive, Nashua, NH 03062

The source code is organized into several main sections:

Common:

- iltypes.h – portable data type definitions
- ilmacro.h – portable macros for common utility functions
- iltime.h – prototypes for date & time conversion functions
- ilutil.h – prototypes for more utility functions
- buffer.c – code for a buffer management mechanism
- iltbl.h – definitions of table info, including field attributes

Engine:

- ilx.h
- ilxapi.h
- loadflds.c – code to assimilate field lists & field map; feeds field into to Synchronizer
- xlate.c – code that drives a synchronization job from start to finish

General Translation Code (used by Synchronizer and by Translators)

- iltr.h
- iltrfn.h – function prototypes
- loadmap.c – get field info ready for use by translators
- charmap.c – map text strings from one character set to another
- fldget.c – get field value from intermediate storage; do field mapping if necessary
- fldput.c – put field value into intermediate storage
- fldtype.c – get info about a field
- sst.c – Section SubType (origin tag) code

General Translation Code for handling Recurring Items and Fanning

- ilrpt.h – defines how recurrence patterns are encoded
- rptnext.c – compute Next Date that belongs to a recurrence pattern
- unfold.c – Fanning code to build an array of most useful dates
- getrep.c – Get recurrence pattern from intermediate storage
- putrep.c – Put recurrence pattern into intermediate storage

Generic Translator Code (used to build specific Translators)

- export.c – top-level for reading records from a Database into Intermediate Storage
- import.c – top-level for writing records into a Database from Intermediate Storage
- ilxtrans.cpp – “C” wrapper for underlying “C++” code
- ciltrans.h – definition of the general Translator class
- ciltrans.cpp – implementation of the general Translator class
- cilrec.h – definition of the ‘Record’ class, for translators
- cilrec.cpp – implementation of the ‘Record’ class for translators
- cilfield.h – definition of Field-handling class for translators
- cilfield.cpp – implementation of field-handling class

Synchronizer

- iltif.h – externally visible definitions
- iltiffn.h – prototypes for all synchronizer entrypoints
- tif.h – internal definitions
- tiffn.h – prototypes for internal functions
- tifsync2.h
- tifrc.h – Ids for all resources
- iltif.rc – resources (strings)
- iltif.cpp – Mostly top-level entrypoints
- iltifa.cpp – More top-level entrypoints and FANNING code

tif.cpp – Core code; phase control, field handling
tifdump.cpp – Code to produce log entries describing workspace contents
tifmex.cpp – Code to Merge Exclusion Lists
tifput.cpp – Code to do the Initial Analysis of records as they enter the workspace
tifsync.cpp – Implements 'CAAR' - the core of synchronization logic
tifsync2.cpp – Record/Outcome Unloading Logic and History File Mgmt
tiftable.cpp – Tables used to determine record outcomes
tifutil.cpp – "Constraint-aware" Hashing and Comparing functions
tifxutil.cpp – more basic functions
tif_ilcr.cpp – Code to present a conflict to a user and handle user's resolution choice

```

#if !defined(__ILTYPES)

/*-----
 * Name:      ILTYPES.H
 * Purpose:   Header file for shared data types
 * Author:    Copyright (c) IntelliLink, 1994-1995
 *-----*/
#define __ILTYPES                // Signal header inclusion

#ifdef ILMAC
    #include "ilrez.h"
#endif

/*-----
 * Constants (formerly "ilconst.h")
 *-----*/
#ifdef __SC
    #undef NULL
    #define NULL            0
#endif

#ifndef NULL
    #ifdef __cplusplus
        #define NULL        0
    #else
        #define NULL        ((void *)0)
    #endif
#endif

#if !defined(FALSE)
    #define FALSE            0
#endif

#if !defined(TRUE)
    #define TRUE             1
#endif

#if !defined(SUCCESS)
    #define SUCCESS          0
#endif

#if !defined(FAILURE)
    #define FAILURE          -1
#endif

/*-----
 * Symbolic constants.
 *-----*/
#define IL_ATTR_READ        1                // Read attribute
#define IL_ATTR_WRITE       2                // Create attribute
#define IL_ATTR_APPEND      3                // Append attribute
#define IL_ATTR_UPDATE      4                // Update attribute

/*-----
 * Limits.
 *-----*/
#ifdef ILWIN
    #ifndef _WIN32
        // 16-bit Windows
        #define MAX_DIR        65            // Max size of directory name
        #define MAX_DRIVE     3             // Max size of drive name
        #define MAX_EXT        5            // Max size of file extension
        #define MAX_FILE_NAME  13           // Max size of file + extension
        #define MAX_FNAME      9            // Max size of file name
        #define MAX_PATH       65           // Max size of path name
    #else
        // 32-bit Windows
        #define MAX_DIR        256           // Max size of directory name
        #define MAX_DRIVE     3             // Max size of drive name
        #define MAX_EXT        256          // Max size of file extension
        #define MAX_FILE_NAME  256          // Max size of file + extension
        #define MAX_FNAME      256          // Max size of file name
        #define MAX_PATH       260          // Max size of path name
    #endif // _WIN32
#endif // ILWIN

```

```

#ifdef ILMAC
#define MAX_DIR 255 // MacIntosh
#define MAX_DRIVE 63 // Max size of directory name
#define MAX_EXT 32 // Max size of drive name
#define MAX_FILE_NAME 95 // Max size of file extension
#define MAX_FNAME 64 // Max size of file + extension
#define MAX_PATH 255 // Max size of file name
#endif

#ifdef ILDOS
#define MAX_DIR 65 // DOS and GEOS, and SYSMGR
#define MAX_DRIVE 3 // Max size of directory name
#define MAX_EXT 5 // Max size of drive name
#define MAX_FILE_NAME 13 // Max size of file extension
#define MAX_FNAME 9 // Max size of file + extension
#define MAX_PATH 65 // Max size of file name
#endif

#ifdef SYSMGR
#define MAX_DIR 65 // HP SYSMGR
#define MAX_DRIVE 3 // Max size of directory name
#define MAX_EXT 5 // Max size of drive name
#define MAX_FILE_NAME 13 // Max size of file extension
#define MAX_FNAME 9 // Max size of file + extension
#define MAX_PATH 65 // Max size of file name
#endif

//----- For "everyone"
#define MAX_APP_DATA_FIELDS 1 //OBSOLETE: use ILTR_EXTRA_FIELDS_ALWAYS
#define MAX_APP_NAME 26 // Max size of application name
#define MAX_FAR_PTRS 100 // Max number of far pointers
#define MAX_FIELD_LEN 512 // Max size of field
#define MAX_FUNCTS 10 // Max number of functions
#define MAX_IF_INCR 8192 // Max IF record increment
#define MAX_IF_ALLOC 36864 // Max size of IF record (32K+4K)
#define MAX_IF_REC_SIZE 4096 // Default size of IF record
#define MAX_LINE_LEN 512 // Max line length
#define MAX_MSG 80 // Max size of text message
#define MAX_REPEAT_FIELDS 2 //OBSOLETE: use ILTR_EXTRA_FIELDS_FOR_REPEAT
#define MAX_USER_FAR_PTRS 20 // Max number of user far ptrs

/*-----
 * Symbols.
 *-----*/
#define IL_CLEAR_PASS -1 // Filter Item clearing value
#define IL_DATE_SIZE 8 // Alpha date field size
#define IL_TIME_SIZE 4 // Alpha time field size

/*-----
 * Basic data types.
 *-----*/
#ifdef ILWIN
#define LITTLE_ENDIAN // Windows
// Bytes are "backwards"
#endif
#ifndef _WIN32
// 16-bit Windows

/*-----
 * Note that WORD and DWORD types are already defined in WINDEF.H
 * and not included here.
 *-----*/
#define DLLEXPORT // Exported DLL function
#define EXP _export // Export keyword for DLL
#define IL_CDECL _cdecl // C declaration type
#define IL_DECL WINAPI // C function declaration
#define IL_DIST _far // Pointer distance
#define IL_HUGE _huge // Huge pointer keyword
#define IL_FILEINFO OFSTRUCT // Windows file info
#define IL_FINDINFO struct _find_t // Windows find info
#define IL_HANDLE HANDLE // Windows memory handle
#define IL_LPHANDLE IL_HANDLE IL_DIST * // Pointer to Window memory handle
#define IL_HFILE int // Windows file handle
#define IL_HFIND long // Windows find handle
#define IL_HINST HINSTANCE // Instance handle
#define IL_HKEY HANDLE // Settings access key handle
#define IL_HWIN HWND // Window handle
#define IL_NULL_HANDLE 0 // Windows null memory handle

```

```

#define IL_NULL_HFILE    -1                // Windows null file handle
#define IL_PCSTR         LPCSTR            // Pointer to read-only string
#define IL_PSTR          LPSTR             // Pointer to string
#define IL_PINT          LPINT             // Pointer to int
#define IL_SPRINTF       sprintf           // String formatting function
#define INT8             char             // 8 bit signed integer
#define INT16            short            // 16 bit signed integer
#define INT32            long             // 32 bit signed integer
#define UINT8            unsigned char    // 8 bit unsigned integer
#define UINT16           unsigned short   // 16 bit unsigned integer
#define UINT32           unsigned long    // 32 bit unsigned integer
#define IL_FILESEP_CH    '\\'            // File name separator character
#define IL_FILESEP_STR   "\\\"           // File name separator string
#define IL_ENDLINE_STR   "\r\n\"         // End of line separator string

typedef INT16 BOOL16;                // BOOLEAN type, ALWAYS 16 bits
typedef INT16 BOOLEAN;              // BOOLEAN type, system dependent

#else                                // 32-bit Windows

/*-----
 * Note that BOOLEAN is already defined in WINNT.H and not included here.
 * WORD and DWORD are standard Windows types defined in WINDEF.H.
 *-----*/
#define DLLEXPORT        __declspec(dllexport) // Exported DLL function
#define EXP              // Export keyword for DLL
#define IL_CDECL         _cdecl              // C declaration type
#define IL_DECL          WINAPI             // C function declaration
#define IL_DIST          // Pointer distance (obsolete)
#define IL_HUGE          // Huge pointer (obsolete)
#define IL_FILEINFO      OFSTRUCT          // Windows file info
#define IL_FINDINFO      struct _finddata_t // Windows find info
#define IL_HANDLE        HGLOBAL           // Windows memory handle
#define IL_LPHANDLE      IL_HANDLE *       // Pointer to Window memory handle
#define IL_HFILE         HFILE             // Windows file handle
#define IL_HFIND         long              // Windows find handle
#define IL_HINST         HINSTANCE         // Instance handle
#define IL_HKEY           HKEY             // Settings access key handle
#define IL_HWIN          HWND             // Window handle
#define IL_NULL_HANDLE   ((HGLOBAL)0)     // Windows null memory handle
#define IL_NULL_HFILE    ((HFILE)-1)      // Windows null file handle
#define IL_PCSTR         LPCSTR            // Pointer to read-only string
#define IL_PSTR          LPSTR             // Pointer to string
#define IL_PINT          LPINT             // Pointer to int
#define IL_SPRINTF       sprintf           // String formatting function
#define INT8             char             // 8 bit signed integer
#define INT16            short            // 16 bit signed integer
#define INT32            long             // 32 bit signed integer
#define UINT8            unsigned char    // 8 bit unsigned integer
#define UINT16           unsigned short   // 16 bit unsigned integer
#define UINT32           unsigned long    // 32 bit unsigned integer
#define IL_FILESEP_CH    '\\'            // File name separator character
#define IL_FILESEP_STR   "\\\"           // File name separator string
#define IL_ENDLINE_STR   "\r\n\"         // End of line separator string

typedef INT16 BOOL16;                // BOOLEAN type, ALWAYS 16 bits

#endif // WIN32
#endif // ILWIN

//----- DOS (and GEOS) types
#ifdef ILDOS                          // DOS
#define LITTLE_ENDIAN      // Bytes are "backwards"
#define DLLEXPORT          // Exported DLL function
#define EXP                // Export keyword
#define IL_CDECL           _cdecl       // C declaration type
#define IL_DECL            _cdecl       // C function declaration
#define IL_DIST            // Pointer distance
#define IL_HUGE            _huge        // Huge pointer keyword
#define IL_FILEINFO        int         // DOS file data
#define IL_FINDINFO        int         // DOS find data
#define IL_HANDLE          int         // DOS memory handle
#define IL_LPHANDLE        IL_HANDLE IL_DIST * // Pointer to DOS memory handle
#define IL_HFILE           int         // DOS file handle
#define IL_HFIND           long        // DOS find handle

```

```

#define IL_HINST          unsigned int          // Instance handle
#define IL_HKEY           unsigned int          // Settings access key handle
#define IL_HWIN          unsigned int          // Window handle
#define IL_NULL_HANDLE    0                    // DOS null memory handle
#define IL_NULL_HFILE     -1                   // DOS null file handle
#define IL_PCSTR          const char IL_DIST * // Pointer to read-only string
#define IL_PSTR           char IL_DIST *       // Pointer to string
#define IL_PINT           int IL_DIST *        // Pointer to int
#define IL_SPRINTF        sprintf              // String formatting function
#define INT8              char                 // 8 bit signed integer
#define INT16             short                // 16 bit signed integer
#define LONG              long                 // 32 bit signed integer
#define INT32             long                 // 32 bit signed integer
#define BYTE              unsigned char        // 8 bit unsigned integer
#define UINT8             unsigned char        // 8 bit unsigned integer
#define UINT              unsigned short       // 16 bit unsigned integer
#define UINT16            unsigned short       // 16 bit unsigned integer
#define UINT32            unsigned long        // 32 bit unsigned integer
#define HFONT             int                  // Font handle
#define FARPROC           void *               // Function pointer def
typedef FARPROC TIMERPROC;                    // Timer function pointer
#define IL_FILESEP_CH     '\\\\'               // File name separator character
#define IL_FILESEP_STR    "\\\\"              // File name separator string
#define IL_ENDLINE_STR    "\r\n"              // End of line separator string

typedef INT16 BOOL16;                          // BOOLEAN type, ALWAYS 16 bits
typedef INT16 BOOLEAN;                          // BOOLEAN type, system dependent
typedef UINT16 WORD;                            // WORD is 16-bit unsigned
typedef UINT32 DWORD;                          // DWORD is 32-bit unsigned

#endif // ILDOS

//----- MacIntosh types
#ifdef ILMAC                                     // Macintosh

//----- Common types used on both Windows and Macintosh
typedef unsigned char    BYTE;                  // 8 bits, unsigned
typedef unsigned short   WORD;                  // 16 bits, unsigned
typedef unsigned long    DWORD;                 // 32 bits, unsigned
typedef unsigned int     UINT;                  // Normally 32 bits, unsigned
typedef signed long      LONG;                  // 32 bits, signed
typedef char*            LPSTR;                 // Pointer to read/write string
typedef const char*      LPCSTR;                // Pointer to read-only string
typedef BYTE*            LPBYTE;                // Pointer to a byte (unsigned)
typedef int*             LPINT;                 // Pointer to a signed integer
typedef WORD*            LPWORD;                // Pointer to an unsigned int
typedef long*            LPLONG;                // Pointer to a signed long
typedef DWORD*           LPDWORD;               // Pointer to an unsigned long
typedef void*            LPVOID;                // Void pointer
typedef int              (* FARPROC) ();        // Function pointer def
typedef FARPROC          TIMERPROC;             // Timer function pointer

//----- Macintosh stores integers "high byte first"
#define BIG_ENDIAN        // Bytes are "forwards"

//----- IntelliLink types for the Macintosh
#define DLLEXPORT          // Exported DLL function
#define EXP                // Export keyword
#define IL_CDECL           // C declaration type
#define IL_DECL            // C function declaration
#define IL_DIST            // Pointer distance
#define IL_HUGE            // Huge pointer keyword
#define IL_FILEINFO        int                  // Mac file data
#define IL_FINDINFO        int                  // Mac find file data
#define IL_HANDLE          LPVOID               // Mac memory handle (really a pointer)
#define IL_LPHANDLE        IL_HANDLE*          // Pointer to Mac memory handle
#define IL_HFILE           int                  // Mac file handle
#define IL_HFIND           int                  // Mac find handle
#define IL_HINST           unsigned int         // Instance handle
#define IL_HKEY            IL_HANDLE            // Settings access key handle
#define IL_HWIN            unsigned int         // Window handle
#define IL_NULL_HANDLE     0                    // Mac null memory handle
#define IL_NULL_HFILE      -1                   // Mac null file handle
#define IL_PCSTR           LPCSTR               // Pointer to read-only string
#define IL_PSTR            LPSTR                // Pointer to string

```

```

#define ILPINT          LPINT          // Pointer to int
#define IL_SPRINTF      sprintf        // String formatting function
#define INT8            char          // 8 bit signed integer
#define INT16           short         // 16 bit signed integer
#define INT32           long          // 32 bit signed integer
#define UINT8           BYTE          // 8 bit unsigned integer
#define UINT16          WORD          // 16 bit unsigned integer
#define UINT32          DWORD         // 32 bit unsigned integer
#define _MAX_DIR        255          // Max directory (folder) name
#define _MAX_DRIVE      255          // Max drive name
#define _MAX_EXT        255          // Max extension name
#define _MAX_FILE       255          // Max file name
#define _MAX_PATH       255          // Max path name
#define HDC int          // Device context
#define HFONT int        // Font handle
#define IL_FILESEP_CH   ':'          // File name separator character
#define IL_FILESEP_STR  ":"          // File name separator string
#define IL_ENDLINE_STR  "\n"        // End of line separator string

typedef INT16 BOOL16;          // BOOLEAN type, ALWAYS 16 bits
typedef INT16 BOOLEAN;        // BOOLEAN type, system dependent

//----- DOS file attributes for the MacIntosh
#define _A_NORMAL       0x00        // Normal file
#define _A_RDONLY       0x01        // Read only file
#define _A_HIDDEN       0x02        // Hidden file
#define _A_SYSTEM       0x04        // System file
#define _A_VOLID        0x08        // Volume ID file
#define _A_SUBDIR       0x10        // Subdirectory
#define _A_ARCH         0x20        // Archive file

//----- File attributes as used in MACFile utilities
#define MACFILE_LOCKED   0x01        // Mac file is locked (read-only)
#define MACFILE_DIRECTORY 0x10      // Mac file is a directory
#define MACFILE_FLOPPY   0x02        // Mac file is root directory of floppy

//----- Throw/catch data structure for the MacIntosh
typedef int CATCHBUF[9];

#endif // ILMAC

//----- System manager types
#ifdef SYSMGR
// System Manager
// Bytes are "backwards"
// Exported DLL function
// Export keyword
// C declaration type
// C function declaration
// Pointer distance
// Huge pointer keyword
// System Manager file data
// System Manager find data
// System Manager memory handle
// Pointer to SM memory handle
// System Manager file handle
// System Manager find handle
// Instance handle
// Settings access key handle
// Window handle
// System Manager null handle
// System Manager null handle
// Pointer to read-only string
// Pointer to string
// Pointer to int
// String formatting function
// 8 bit signed integer
// 16 bit signed integer
// 32 bit signed integer
// 32 bit signed integer
// 8 bit unsigned integer
// 8 bit unsigned integer
// 16 bit unsigned integer
// 16 bit unsigned integer
// 32 bit unsigned integer
// 32 bit unsigned integer
// Font handle

```



```

#define FARPROC void *           // Function pointer def
typedef FARPROC TIMERPROC;      // Timer function pointer
#define IL_FILESEP_CH '\\\\'     // File name separator character
#define IL_FILESEP_STR "\\\"    // File name separator string
#define IL_ENDLINE_STR "\r\n"   // End of line separator string

typedef INT16 BOOL16;           // BOOLEAN type, ALWAYS 16 bits
typedef INT16 BOOLEAN;          // BOOLEAN type, system dependent
typedef UINT16 WORD;            // WORD is 16-bit unsigned
typedef UINT32 DWORD;           // DWORD is 32-bit unsigned

#endif // SYSMGR

//----- Make sure other common types are defined
#ifdef ILWIN
#define IL_PANY LPVOID           // Pointer to void type
#else
#define IL_PANY void IL_DIST *   // Pointer to void type
#endif // ILWIN

/*-----
 * Definition of alpha Date type (used in intermediate file)
-----*/

typedef struct
{
    union
    {
        struct
        {
            char Year[4];         // YYYY
            char Month[2];        // MM
            char Day[2];          // DD
        } str;
        char String[IL_DATE_SIZE+1];
    } u;
} IL_DATE;

/*-----
 * Definition of Time type.
-----*/

typedef struct
{
    union
    {
        struct
        {
            char Hour[2];         // HH (00-23)
            char Minute[2];       // MM (00-59)
        } str;
        char String[IL_TIME_SIZE+1];
    } u;
} IL_TIME;

/*-----
 * ILBASE definitions
-----
 *
 * The following definitions are used to declare functions that we
 * may want to put into an "ILBASE" DLL rather than statically
 * linking them into each translator.
 *
 * Give ILBASEFN_PREFIX and ILBASEFN_SUFFIX NULL definitions to
 * leave such functions statically linked.
-----*/

#ifdef ILWIN_USING_ILBASE16
#define ILBASEFN_PREFIX
#define ILBASEFN_SUFFIX EXP
#else
#ifdef ILWIN_USING_ILBASE32
#define ILBASEFN_PREFIX DLLEXPORT
#define ILBASEFN_SUFFIX
#else
#define ILBASEFN_PREFIX

```

```
#define ILBASEFN_SUFFIX
#endif
#endif

#define ILBASEFN_BOOL16 ILBASEFN_PREFIX BOOL16 IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_BOOLEAN ILBASEFN_PREFIX BOOLEAN IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_HILIF ILBASEFN_PREFIX HILIF IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_IL_PSTR ILBASEFN_PREFIX IL_PSTR IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_ILTB_ID ILBASEFN_PREFIX ILTB_ID IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_ILTB_PCONFIG \
    ILBASEFN_PREFIX ILTB_PCONFIG IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_ILTB_PMAP \
    ILBASEFN_PREFIX ILTB_PMAP IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_int ILBASEFN_PREFIX int IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_INT32 ILBASEFN_PREFIX INT32 IL_DECL ILBASEFN_SUFFIX
#define ILBASEFN_void ILBASEFN_PREFIX void IL_DECL ILBASEFN_SUFFIX

#endif // __ILTYPES
```

```

#ifndef __ILMACRO

/*-----
 * Name:      ILMACRO.H
 * Purpose: Header file for IntelliLink macros
 * Author: Copyright (c) IntelliLink, 1994
 *-----*/
#define __ILMACRO // Signal header inclusion

/*-----
 * Required headers.
 *-----*/
#ifdef ILWIN
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#endif

#ifdef ILDOS
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#endif

#ifdef SYSMGR
#include "malloc.h"
#include "interfac.h"
#include "fileio.h"
#include <io.h>
#endif

#ifdef ILMAC
#ifdef __MWERKS__
#include <unix.h>
#else
#include <fcntl.h>
#endif
#include <files.h>
#include <types.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <TextUtils.h>
#endif

#include "iltypes.h"

/*-----
 * Windows macros -- 16 and 32-bit versions.
 *-----*/
#ifdef ILWIN

/*-----
 * IntelliLink convention: allocation of zero-length blocks is a no-no.
 *-----
 * All programs should refrain from calling IL_ALLOC_MEM with nMem=0.
 * The ILWIN and ILDOS versions of IL_ALLOC_MEM behave differently when
 * asked to allocate zero bytes.
 * The ILWIN version gives you a NULL pointer (same result that you
 * get when allocation fails due to memory shortfall!!).
 * The ILDOS version, on the other hand, returns a valid pointer to a
 * zero-length item in the heap.
 * Previous to 7/7/94 there was a bug whereby zero-length ILWIN
 * IL_ALLOC_MEM requests led to consumption of precious windows handles!!
 *-----*/

//----- Define the 16-bit macros for ILWIN
#endif

```

```

#define IL_ALLOC_MEM(nMem, hMem, pMem) (IL_PSTR) \
    pMem = ILUT_MemAlloc (nMem, &(hMem))

#define IL_FREE_MEM(hMem, pMem) ILUT_MemFree(hMem, pMem)

#define IL_FREE_MEM_AND_ZERO_HANDLE(hMem, pMem) \
    ( IL_FREE_MEM(hMem, pMem); hMem = IL_NULL_HANDLE; )

#define IL_REALLOC_MEM(nMem, hMem, pMem) (IL_PSTR) \
    pMem = ILUT_MemReAlloc (nMem, &(hMem), pMem)

#define IL_SYNC_MEM(hMem, pMem)

// ----- Variations of IL_ALLOC_MEM usable by C and C++

#define IL_ALLOC(nMem, hMem) ILUT_MemAlloc (nMem, &(hMem))

#define IL_FREE IL_FREE_MEM

#define IL_FREE_AND_ZERO(hMem, pMem) \
    ( IL_FREE(hMem, pMem); hMem = IL_NULL_HANDLE; pMem = NULL; )

#define IL_REALLOC(nMem, hMem, pMem) ILUT_MemReAlloc (nMem, &(hMem), pMem)

#define IL_OPEN(name, attr, hFile, info, error) \
    ( error = WinFile_Open(name, attr, &info, &hFile, __FILE__, __LINE__); )

#define IL_READ(hFile, buffer, count, countRead, error) \
    ( error = _lread (hFile, (IL_PSTR) buffer, (WORD) count); \
      countRead = (error == -1) ? 0 : error; \
      error = (error == -1) ? -1 : 0; )

#define IL_WRITE(hFile, buffer, count, error) \
    ( error = _lwrite (hFile, (IL_PSTR) buffer, (WORD) count); \
      error = ((error == -1) || ((int)(error) != (int)(count))) ? -1 : 0; )

#define IL_CLOSE(hFile, error) \
    error = WinFile_Close (hFile, __FILE__, __LINE__)

#define IL_REMOVE(name, info, error) \
    ( error = OpenFile (name, (LPOFSTRUCT) (&info), OF_DELETE); \
      error = (error == HFILE_ERROR) ? -1 : 0; )

#define IL_GET_FILE_SIZE(hFile, lCount, nError) \
    ( nError = (lCount = ILUT_GetFileSize (hFile)) == -1 ? -1 : 0; )

#define IL_SEEK(hFile, value, mode, error) \
    ( if (_llseek (hFile, (LONG) value, mode) == -1) \
      error = -1; \
      else error = 0; )

#define IL_SEEK_BEGIN        SEEK_SET
#define IL_SEEK_CURRENT      SEEK_CUR
#define IL_SEEK_END          SEEK_END
#define IL_TELL(hFile, lPos) \
    ( lPos = _llseek (hFile, 0L, 1); )

#define IL_MEMCHR            _fmemchr
#define IL_MEMCMP            _fmemcmp
#define IL_MEMCPY            _fmemcpy
#define IL_MEMMOVE           _fmemmove
#define IL_MEMSET            _fmemset
#define IL_STRCAT            _fstrcat
#define IL_STRCHR            _fstrchr
#define IL_STRCMP(x, lx, y, ly) _fstrcmp(x, y)
#define IL_STRCOMP           _fstrcmp
#define IL_STRICMP           _fstricmp
#define IL_STRCPY            _fstrcpy
#define IL_STRCSPN           _fstrcspn
#define IL_STRLEN            _fstrlen
#define IL_STRLWR            AnsiLower
#define IL_STRNCAT           _fstrncat
#define IL_STRNCMP           _fstrncmp
#define IL_STRNCPY           _fstrncpy
#define IL_STRNICMP          _fstrnicmp
#define IL_STRRCHR           _fstrrchr
#define IL_STRSPN            _fstrspn
#define IL_STRSTR            _fstrstr
#define IL_STRTOK            _fstrtok

```

```

#define IL_STRDUP(s,r)          r = _fstrdup(s)
#define IL_STRUPR              AnsiUpper

#define IL_SPLITPATH           IL_splitpath
#define IL_MAKEPATH            IL_makepath
#define IL_GETTEMPFILENAME     ILUT_GetTempFileName

#define IL_CHSIZE              chsize
#define IL_DOES_EXIST(filename) ((access(filename,00)==0) ? 1:0)
#define IL_EXISTS(fileName, flag) \
    { flag = access (fileName, 00); flag = (flag == 0) ? 1 : 0; }
#define IL_RENAME(old, new, error) \
    (error = rename (old, new))
#define IL_BEEP()              MessageBeep(0)
#define IL_CHDIR(newdir, error) \
    (error = _chdir (newdir))
#define IL_GETCURDIR(pDir, nLen, nRc) \
    { if (getcwd (pDir, nLen)) \
        nRc = 0; \
      else \
        nRc = -1; \
    }
#define IL_GETDRIVE(drive) \
    drive[0] = (char)(_getdrive() + 'A' - 1)
#define IL_GETDIR(drive, dir, len, error) \
    if (_getcwd (drive - 'A' + 1, dir, len)) \
        error = 0; \
    else \
        error = -1;
#define IL_GETPID(nPid) \
    (nPid = getpid ())
#define IL_MKDIR(newdir, error) \
    (error = _mkdir (newdir))
#define IL_RMDIR(newdir, error) \
    (error = _rmdir (newdir))
#define IL_SETDIR(pDir) \
    _chdir (pDir);
#define IL_SETDRIVE(drive) \
    _chdrive(drive - 'A' + 1)

#define LoadDll(name, hDll) \
    {if ((UINT)(hDll = LoadLibrary (name)) < 32) hDll = NULL;}
#define UnloadDll(hDll) \
    {if ((UINT)hDll >= 32) FreeLibrary (hDll);}

//----- File pattern matching/enumeration
#define IL_FINDCLOSE(h, rc) { rc = 0; h = 0; }
#define IL_FINDFIRST(nm,at,st,h,rc) rc = _dos_findfirst ((nm),(at),&(st))
#define IL_FINDNEXT(h,st,rc) rc = _dos_findnext (&(st))

//----- do pointer subtraction and get INT32 result. We need this macro
//----- to avoid disaster when difference is > 32767, because
//----- PTRDIFF_T is short for this environment.
#define IL_PTRDIFF(a,b) ( ((INT32) b) - ((INT32) a) )

//----- do pointer subtraction for IL_HUGE pointers. This macro generates
//----- a call to the runtime function "_aFahdiff"
#define IL_HUGE_PTRDIFF(a,b) ( ((char IL_HUGE *) b) - ((char IL_HUGE *) a) )

/*-----
 * Use IL_HANDLE_PADDING after any IL_HANDLE member of any structure that
 * needs to be size-invariant from one platform to another. For each plat-
 * form sizeof(IL_HANDLE) + sizeof(IL_HANDLE_PADDING) = 4 bytes.
 *-----*/
#define IL_HANDLE_PADDING(_name) INT16 _name;

//----- Compatibility macros (may not defined in windowsx.h)
#ifndef GET_WM_COMMAND_CMD
    #define GET_WM_COMMAND_CMD(wp, lp)          HIWORD(lp)
    #define GET_WM_COMMAND_ID(wp, lp)           (wp)
    #define GET_WM_COMMAND_HWND(wp, lp)         (HWND) LOWORD(lp)
#endif
#endif

#else // _WIN32 IS defined -- The Win32 versions of the macros follow

/*-----

```

```

* Define the 32-bit versions of the ILWIN macros here. The current
* set of macros was copied from the 16-bit section and modified as
* specific 32-bit issues were encountered in the porting process. At
* the preset time, the following macros are different from their
* 16-bit counterparts:
*
*   IL_MEM...      // Use standard, not _f(ar) functions
*   IL_STR...      // Use standard, not _f(ar) functions
*   IL_EXISTS      // Use _access instead of access
*
* Please make every effort to keep this list up-to-date as additional
* 32-bit-specific macros are added
*-----*/

#define IL_ALLOC_MEM(nMem, hMem, pMem) (IL_PSTR) \
    pMem = ILUT_MemAlloc (nMem, &(hMem))

#define IL_FREE_MEM(hMem, pMem) ILUT_MemFree(hMem, pMem)

#define IL_FREE_MEM_AND_ZERO_HANDLE(hMem, pMem) \
    { IL_FREE_MEM(hMem, pMem); hMem = IL_NULL_HANDLE; }

#define IL_REALLOC_MEM(nMem, hMem, pMem) (IL_PSTR) \
    pMem = ILUT_MemReAlloc (nMem, &(hMem), pMem)

#define IL_SYNC_MEM(hMem, pMem)

// ----- Variations of IL_ALLOC_MEM usable by C and C++

#define IL_ALLOC(nMem, hMem) ILUT_MemAlloc (nMem, &(hMem))

#define IL_FREE IL_FREE_MEM

#define IL_FREE_AND_ZERO(hMem, pMem) \
    { IL_FREE(hMem, pMem); hMem = IL_NULL_HANDLE; pMem = NULL; }

#define IL_REALLOC(nMem, hMem, pMem) ILUT_MemReAlloc (nMem, &(hMem), pMem)

#define IL_OPEN(name, attr, hFile, info, error) \
    { error = WinFile_Open(name, attr, &info, &hFile, __FILE__, __LINE__); }

#define IL_READ(hFile, buffer, count, countRead, error) \
    { error = _lread (hFile, (IL_PSTR) buffer, (UINT) count); \
      countRead = (error == -1) ? 0 : error; \
      error = (error == -1) ? -1 : 0; }

#define IL_WRITE(hFile, buffer, count, error) \
    { error = _lwrite (hFile, (IL_PSTR) buffer, (UINT) count); \
      error = ((error == -1) || ((int)(error) != (int)(count))) ? -1 : 0; }

#define IL_CLOSE(hFile, error) \
    error = WinFile_Close (hFile, __FILE__, __LINE__)

#define IL_REMOVE(name, info, error) \
    { error = DeleteFile (name) ? 0 : -1; info; }

#define IL_GET_FILE_SIZE(hFile, lCount, nError) \
    { nError = (lCount = ILUT_GetFileSize (hFile)) == -1 ? -1 : 0; }

#define IL_SEEK(hFile, value, mode, error) \
    { if (_llseek (hFile, (LONG) value, mode) == -1) \
      error = -1; \
      else error = 0; }

#define IL_SEEK_BEGIN      SEEK_SET
#define IL_SEEK_CURRENT    SEEK_CUR
#define IL_SEEK_END        SEEK_END

#define IL_TELL(hFile, lPos) \
    { lPos = _llseek (hFile, 0L, 1); }

#define IL_MEMCHR          memchr
#define IL_MEMCMP          memcmp
#define IL_MEMCPY          memcpy
#define IL_MEMMOVE         memmove
#define IL_MEMSET          memset
#define IL_STRCAT          strcat
#define IL_STRCHR          strchr
#define IL_STRCMP(x,lx,y,ly) strcmp(x,y)
#define IL_STRCOMP         strcmp
#define IL_STRICMP         _stricmp

```



```

#define IL_STRCPY      strcpy
#define IL_STRCSPN     strcspn
#define IL_STRLEN      strlen
#define IL_STRLWR      CharLower
#define IL_STRNCAT     strncat
#define IL_STRNCMP     strncmp
#define IL_STRNCPY     strncpy
#define IL_STRNICMP    strnicmp
#define IL_STRRCHR     strrchr
#define IL_STRSPN      strspn
#define IL_STRSTR      strstr
#define IL_STRTOK      strtok
#define IL_STRDUP(s,r) r = _strdup(s)
#define IL_STRUPR      CharUpper

#define IL_SPLITPATH   _splitpath
#define IL_MAKEPATH    _makepath
#define IL_GETTEMPFILENAME ILUT_GetTempFileName

#define IL_CHSIZE      _chsize
#define IL_DOES_EXIST(filename) (( _access(filename,00)==0) ? 1:0)
#define IL_EXISTS(fileName, flag) \
    { flag = _access (fileName, 00); flag = (flag == 0) ? 1 : 0; }
#define IL_RENAME(old, new, error) \
    (error = rename (old, new))
#define IL_BEEP()      MessageBeep(0)
#define IL_CHDIR(newdir, error) \
    (error = _chdir (newdir))
#define IL_GETCURDIR(pDir, nLen, nRc) \
    { if ( _getcwd (pDir, nLen)) \
        nRc = 0; \
      else \
        nRc = -1; \
    }
#define IL_GETDRIVE(drive) \
    drive[0] = (char)( _getdrive() + 'A' - 1)
#define IL_GETDIR(drive, dir, len, error) \
    if ( _getcwd (drive - 'A' + 1, dir, len)) \
        error = 0; \
    else \
        error = -1;
#define IL_GETPID(nPid) \
    (nPid = _getpid ())
#define IL_MKDIR(newdir, error) \
    (error = _mkdir (newdir))
#define IL_RMDIR(newdir, error) \
    (error = _rmdir (newdir))
#define IL_SETDIR(pDir) \
    _chdir (pDir);
#define IL_SETDRIVE(drive) \
    _chdrive(drive - 'A' + 1)

#define LoadDll(name, hDll) \
    hDll = LoadLibraryEx (name, NULL, LOAD_WITH_ALTERED_SEARCH_PATH)
#define UnloadDll(hDll)      {if (hDll != NULL) FreeLibrary (hDll);}

//----- File pattern matching/enumeration
#define IL_FINDCLOSE(h,rc) rc = _findclose (h)
#define IL_FINDFIRST(nm,at,st,h,rc) { \
    (st).attrib = (at); \
    (h) = _findfirst ((nm), &(st)); \
    (rc) = ((h) == -1) ? 0 : -1; \
}
#define IL_FINDNEXT(h,st,rc) rc = _findnext ((h), &(st))

//----- do pointer subtraction and get INT32 result. This is trivial
//----- here because PTRDIFF_T is long for this environment.
#define IL_PTRDIFF(a,b) (b-a)
#define IL_HUGE_PTRDIFF(a,b) (b-a)

/*-----
 * Use IL_HANDLE_PADDING after any IL_HANDLE member of any structure that
 * needs to be size-invariant from one platform to another. For each plat-
 * form sizeof(IL_HANDLE) + sizeof(IL_HANDLE_PADDING) = 4 bytes.
 *-----*/

```

```

#define IL_HANDLE_PADDING(_name)      // no padding needed for WIN32

#endif // end of Win32 macros

#endif // end of ILWIN (16 and 32-bit) macros

/*-----
* DOS macros.
*-----*/
#ifdef ILDOS
#define IL_ALLOC_MEM(nMem, hMem, pMem) \
    { hMem = 0; pMem = (nMem > 0xffff) ? NULL : malloc ((unsigned) nMem); }
#define IL_FREE_MEM(hMem, pMem) \
    if (pMem) free (pMem)
#define IL_REALLOC_MEM(nMem, hMem, pMem) \
    { hMem = 0; \
      pMem = (nMem > 0xffff) ? NULL : realloc (pMem, (unsigned) nMem); }
#define IL_SYNC_MEM(hMem, pMem)

// ----- Variations of IL_ALLOC_MEM usable by C and C++
#define IL_ALLOC(nMem, hMem) \
    ((hMem = 0), ((nMem > 0xffff) ? NULL : malloc ((unsigned) nMem)))

#define IL_FREE IL_FREE_MEM

#define IL_FREE_AND_ZERO(hMem, pMem) \
    { IL_FREE(hMem, pMem); hMem = IL_NULL_HANDLE; pMem = NULL; }

#define IL_REALLOC(nMem, hMem, pMem) \
    ((hMem = 0), ((nMem > 0xffff) ? NULL : realloc (pMem, (unsigned) nMem)))

#define IL_OPEN(fileName, attr, hFile, info, error) \
    { switch (attr) { \
      case IL_ATTR_READ: \
        hFile = open (fileName, O_BINARY|O_RDONLY, S_IREAD); \
        break; \
      case IL_ATTR_WRITE: \
        hFile = open (fileName, O_WRONLY|O_CREAT|O_BINARY|O_TRUNC,S_IWRITE); \
        break; \
      case IL_ATTR_APPEND: \
        hFile = open (fileName, O_BINARY|O_WRONLY|O_APPEND, S_IWRITE); \
        break; \
      case IL_ATTR_UPDATE: \
        hFile = open (fileName, O_BINARY|O_RDWR, S_IWRITE); \
        break; \
    } \
    info = 0; error = (hFile == -1) ? -1 : 0; }
#define IL_READ(hFile, buffer, count, countRead, error) \
    { error = read (hFile, buffer, count); \
      countRead = (error == -1) ? 0 : error; \
      error = (error == -1) ? -1 : 0; }
#define IL_WRITE(hFile, buffer, count, error) \
    { error = write (hFile, buffer, count); \
      error = ((error == -1) || ((int)(error) != (int)(count))) ? -1 : 0; }
#define IL_CLOSE(hFile, error) \
    error = close (hFile)
#define IL_REMOVE(fileName, info, error) \
    { error = remove (fileName); info = 0; }
#define IL_GET_FILE_SIZE(hFile, count, error) \
    { count = filelength (hFile); error = (count == -1) ? -1 : 0; }
#define IL_SEEK(hFile, value, mode, error) \
    { if (lseek (hFile, value, mode) == -1L) \
      error = -1; \
    else error = 0; }
#define IL_SEEK_BEGIN      SEEK_SET
#define IL_SEEK_CURRENT    SEEK_CUR
#define IL_SEEK_END        SEEK_END
#define IL_TELL(hFile, val) val = tell (hFile)
#define IL_MEMCHR          memchr
#define IL_MEMCMP          memcmp
#define IL_MEMCPY          memcpy
#define IL_MEMMOVE         memmove
#define IL_MEMSET          memset
#define IL_STRCAT          strcat
#define IL_STRCHR          strchr

```

```

#define IL_STRCMP(x,lx,y,ly)  strcmp(x,y)
#define IL_STRCOMP          strcmp
#define IL_STRICMP          stricmp
#define IL_STRCPY          strcpy
#define IL_STRCSPN          strcspn
#define IL_STRLEN          strlen
#define IL_STRLWR          strlwr
#define IL_STRNCAT          strncat
#define IL_STRNCMP          strncmp
#define IL_STRNCPY          strncpy
#define IL_STRNICMP          strnicmp
#define IL_STRRCHR          strrchr
#define IL_STRSPN          strspn
#define IL_STRSTR          strstr
#define IL_STRTOK          strtok
#define IL_STRDUP(s,r)      r = strdup(s)
#define IL_STRUPR          strupr

#define IL_SPLITPATH          IL_splitpath
#define IL_MAKEPATH          IL_makepath
#define IL_GETTEMPFILENAME    ILUT_GetTempFileName

#define IL_CHSIZE            chsize
#define IL_DOES_EXIST(filename) ((access(filename,0)==0) ? 1:0)
#define IL_EXISTS(fileName, flag) \
    { flag = access (fileName, 00); flag = (flag == 0) ? 1 : 0; }
#define IL_RENAME(old, new, error) \
    (error = rename (old, new))
#define IL_BEEP()  printf ("\a")
#define IL_CHDIR(newdir, error) \
    (error = _chdir (newdir))
#define IL_GETCURDIR(pDir, nLen, nRc) \
    ( if (getcwd (pDir, nLen)) \
      nRc = 0; \
      else \
      nRc = -1; \
    )
#define IL_GETDRIVE(drive) \
    drive[0] = (char)(_getdrive() + 'A' - 1)
#define IL_GETDIR(drive, dir, len, error) \
    if (_getdcwd (drive - 'A' + 1, dir, len)) \
        error = 0; \
    else \
        error = -1;
#define IL_GETPID(nPid) \
    (nPid = getpid ())
#define IL_MKDIR(newdir, error) \
    (error = _mkdir (newdir))
#define IL_RMDIR(newdir, error) \
    (error = _rmdir (newdir))
#define IL_SETDIR(pDir) \
    chdir (pDir);
#define IL_SETDRIVE(drive) \
    _chdrive(drive - 'A' + 1)

//----- File pattern matching/enumeration
#define IL_FINDCLOSE(h, rc)  rc = 0
#define IL_FINDFIRST(nm,at,st,h,rc) rc = _dos_findfirst ((nm),(at),&(st))
#define IL_FINDNEXT(h,st,rc) rc = _dos_findnext (&(st))

//----- do pointer subtraction and get INT32 result.  We need this macro
//----- to avoid disaster when difference is > 32767, because
//----- PTRDIFF_T is short for this environment.
//----- WARNING: do NOT use this for HUGE pointers
#define IL_PTRDIFF(a,b) ( ((INT32) b) - ((INT32) a) )

//----- do pointer subtraction for IL_HUGE pointers.  This macro generates
//----- a call to the runtime function "__afahdiff"
#define IL_HUGE_PTRDIFF(a,b) ( ((char IL_HUGE *) b) - ((char IL_HUGE *) a) )

/*-----
 * Use IL_HANDLE_PADDING after any IL_HANDLE member of any structure that
 * needs to be size-invariant from one platform to another.  For each plat-
 * form sizeof(IL_HANDLE) + sizeof(IL_HANDLE_PADDING) = 4 bytes.
 *-----*/

```

```

#define IL_HANDLE_PADDING(_name)  INT16 _name;

#endif

/*-----
 * System Manager macros.
 *-----*/
#ifdef SYSMGR
#define IL_ALLOC_MEM(nMem, hMem, pMem) \
    { hMem = 0; pMem = _nmalloc ((unsigned) nMem); }

#define IL_FREE_MEM(hMem, pMem) \
    if (pMem) _nfree (pMem)

#define IL_FREE_MEM_AND_ZERO_HANDLE(hMem, pMem) \
    { IL_FREE_MEM(hMem, pMem); hMem = IL_NULL_HANDLE; }

#define IL_REALLOC_MEM(nMem, hMem, pMem) \
    { hMem = 0; pMem = _nrealloc (pMem, (unsigned) nMem); }

// ----- Variations of IL_ALLOC_MEM usable by C and C++
#define IL_ALLOC(nMem, hMem) \
    ((hMem = 0), ((nMem > 0xffff) ? NULL : _nmalloc ((unsigned) nMem)))

#define IL_FREE IL_FREE_MEM

#define IL_FREE_AND_ZERO(hMem, pMem) \
    { IL_FREE(hMem, pMem); hMem = IL_NULL_HANDLE; pMem = NULL; }

#define IL_REALLOC(nMem, hMem, pMem) \
    ((hMem = 0), ((nMem > 0xffff) ? NULL : _nrealloc (pMem, (unsigned) nMem)))

#define IL_SYNC_MEM(hMem, pMem)

#define IL_OPEN(fileName, attr, hFile, info, error) \
    switch (attr) { \
    case IL_ATTR_READ: \
        error = m_openro (&(hFile), fileName, _fstrlen (fileName), 0, 0); \
        break; \
    case IL_ATTR_WRITE: \
        error = m_fcreat (&(hFile), fileName, _fstrlen (fileName), 0, 0); \
        break; \
    case IL_ATTR_APPEND: \
        error = m_open (&(hFile), fileName, _fstrlen (fileName), 0, 0); \
        info = 0; m_seek (&(hFile), seek_end, (long) 0); \
        break; \
    case IL_ATTR_UPDATE: \
        error = m_open (&(hFile), fileName, _fstrlen (fileName), 0, 0); \
        info = 0; \
        break; \
    }

#define IL_READ(hFile, buffer, count, countRead, error) \
    error = m_read (&(hFile), buffer, count, &(countRead))
#define IL_WRITE(hFile, buffer, count, error) \
    error = m_write (&(hFile), buffer, count)
#define IL_CLOSE(hFile, error) \
    error = m_close (&(hFile))
#define IL_REMOVE(fileName, info, error) \
    { error = m_delete (fileName, _fstrlen (fileName), 0); info = 0; }
#define IL_GET_FILE_SIZE(hFile, count, error) \
    { m_seek (&(hFile), seek_end, (long) 0); \
    error = m_tell (&(hFile), &(count)); \
    error = (error != 0) ? -1 : 0; }
#define IL_SEEK(hFile, value, mode, error) \
    error = m_seek (&(hFile), mode, value)
#define IL_SEEK_BEGIN    seek_beginning
#define IL_SEEK_CURRENT  seek_current
#define IL_SEEK_END      seek_end
#define IL_TELL(hFile, val) m_tell (&(hFile), &(val))
#define IL_MEMCHR        _fmemchr
#define IL_MEMCMP        _fmemcmp
#define IL_MEMCPY        _fmemcpy
#define IL_MEMMOVE       _fmemmove
#define IL_MEMSET        _fmemset
#define IL_STRCAT        _fstrcat

```

```

#define IL_STRCHR          _fstrchr
// System manager uses inverse semantics for return value;
// hence the negation of the result
#define IL_STRCMP(x,lx,y,ly) -(m_col_cpstr(x,lx,y,ly))
#define IL_STRICMP          _fstricmp
#define IL_STRCPY          _fstrncpy
#define IL_STRCSPN          _fstrcspn
#define IL_STRLen          _fstrlen
#define IL_STRLWR          _fstrlwr
#define IL_STRNCAT          _fstrncat
#define IL_STRNCMP          _fstrncmp
#define IL_STRNICMP          _fstrnicmp
#define IL_STRNCPY          _fstrncpy
#define IL_STRRCHR          _fstrrchr
#define IL_STRSPN          _fstrspn
#define IL_STRSTR          _fstrstr
#define IL_STRTOK          _fstrtok
#define IL_STRDUP(s,r)      r = _fstrdup(s)
#define IL_STRUPR          _fstrupr

#define IL_SPLITPATH        IL_splitpath
#define IL_MAKEPATH          IL_makepath
#define IL_GETTEMPFILENAME  ILUT_GetTempFileName

#define IL_CHSIZE            //***** NOT IMPLEMENTED!
#define IL_DOES_EXIST(filename) (IL_Exists(filename))
#define IL_EXISTS(fileName, flag) \
    { m_ident (fileName, _fstrlen (fileName), 0, &flag); \
      flag = (flag == 0) ? 0 : 1; }
#define IL_RENAME(old, new, error) \
    (error = m_rename (old, _fstrlen (old), 0, new, _fstrlen (new), 0))
#define IL_BEEP()          m_beep()
#define IL_CHDIR(newdir, error) \
    (error = m_setdir(newdir, _fstrlen (newdir)))
#define IL_GETCURDIR(pDir, nLen, nRc) \
    nRc = -1;                //***** NOT IMPLEMENTED!
#define IL_GETDRIVE(drive) \
    m_getdrv(drive)
#define IL_GETDIR(drive, dir, len, error) \
    m_getdir(drive, dir, len)
#define IL_GETPID(nPid)      //***** NOT IMPLEMENTED!
#define IL_MKDIR(newdir, error) \
    (error = m_mkdir(newdir, _fstrlen (newdir), 0))
#define IL_RMDIR(newdir, error) \
    (error = m_rmdir(newdir, _fstrlen (newdir), 0))
#define IL_SETDIR(pDir)      //***** NOT IMPLEMENTED!
#define IL_SETDRIVE(drive) \
    m_setdrv(drive)

//----- File pattern matching/enumeration - NOT IMPLEMENTED!!!!
#define IL_FINDCLOSE(h,rc)
#define IL_FINDFIRST(nm,at,st,h,rc)
#define IL_FINDNEXT(h,st,rc)

//----- do pointer subtraction and get INT32 result. We need this macro
//----- to avoid disaster when difference is > 32767, because
//----- PTRDIFF_T is short for this environment.
#define IL_PTRDIFF(a,b) ( ((INT32) b) - ((INT32) a) )

//----- do pointer subtraction for IL_HUGE pointers. This macro generates
//----- a call to the runtime function "_aFahdiff"
#define IL_HUGE_PTRDIFF(a,b) ( ((char IL_HUGE *) b) - ((char IL_HUGE *) a) )

/*-----
 * Use IL_HANDLE_PADDING after any IL_HANDLE member of any structure that
 * needs to be size-invariant from one platform to another. For each plat-
 * form sizeof(IL_HANDLE) + sizeof(IL_HANDLE_PADDING) = 4 bytes.
 *-----*/
#define IL_HANDLE_PADDING(_name) INT16 _name;

#endif

/*-----
 * Macintosh macros.
 *-----*/

```

```

#ifndef ILMAC

#define IL_ALLOC_MEM(nMem, hMem, pMem) \
    pMem = (IL_PANY)ILUT_MemAlloc (nMem, &(hMem))
#define IL_FREE_MEM(hMem, pMem) ILUT_MemFree(hMem, pMem)

#define IL_FREE_MEM_AND_ZERO_HANDLE(hMem, pMem) \
    ( IL_FREE_MEM(hMem, pMem); hMem = IL_NULL_HANDLE; )

#define IL_REALLOC_MEM(nMem, hMem, pMem) \
    pMem = (IL_PANY)ILUT_MemReAlloc (nMem, &(hMem), pMem)

#define IL_SYNC_MEM(hMem, pMem)

// ----- Variations of IL_ALLOC_MEM usable by C and C++

#define IL_ALLOC(nMem, hMem) ILUT_MemAlloc (nMem, &(hMem))

#define IL_FREE IL_FREE_MEM

#define IL_FREE_AND_ZERO(hMem, pMem) \
    ( IL_FREE(hMem, pMem); hMem = IL_NULL_HANDLE; pMem = NULL; )

#define IL_REALLOC(nMem, hMem, pMem) ILUT_MemReAlloc (nMem, &(hMem), pMem)

#if 0 //----- THE following code can be deleted once switch to MEMUTIL is tested.
//----- All ILMAC memory allocation is done through MACALLOC.C
#define IL_ALLOC_MEM(nMem, hMem, pMem) \
    pMem = MacMem_Lock (hMem = MacMem_Alloc ((UINT32) nMem))
\

#define IL_FREE_MEM(hMem, pMem) \
    MacMem_Free (hMem)

#define IL_FREE_MEM_AND_ZERO_HANDLE(hMem, pMem) \
    ( IL_FREE_MEM(hMem, pMem); hMem = IL_NULL_HANDLE; )

#define IL_REALLOC_MEM(nMem, hMem, pMem) \
    pMem = MacMem_Lock (hMem = MacMem_Realloc ((UINT32) nMem, hMem))
\

#define IL_SYNC_MEM(hMem, pMem)

// ----- Variations of IL_ALLOC_MEM usable by C and C++
#define IL_ALLOC(nMem, hMem) \
    MacMem_Lock (hMem = MacMem_Alloc ((UINT32) nMem))
\

#define IL_FREE IL_FREE_MEM

#define IL_FREE_AND_ZERO(hMem, pMem) \
    ( IL_FREE(hMem, pMem); hMem = IL_NULL_HANDLE; pMem = NULL; )

#define IL_REALLOC(nMem, hMem, pMem) \
    MacMem_Lock (hMem = MacMem_Realloc ((UINT32) nMem, hMem))
\
#endif

#define IL_OPEN(fileName, attr, hFile, info, error) \
    {error = MACFile_Open(fileName, attr, &hFile); info = 0;}
#define IL_READ(hFile, buffer, count, countRead, error) \
    error = MACFile_Read(hFile, (char *)buffer, (int) count, \
        (int *)&countRead)
#define IL_WRITE(hFile, buffer, count, error) \
    error = MACFile_Write(hFile, (char *)buffer, count)
#define IL_CLOSE(hFile, error) \
    error = MACFile_Close (hFile)
#define IL_REMOVE(fileName, info, error) \
    { error = remove (fileName); info = 0; }
#define IL_GET_FILE_SIZE(hFile, lCount, nError) \
    { nError = (lCount = MACFile_Size (hFile)) == -1 ? -1 : 0; }
#define IL_SEEK(hFile, value, mode, error) \
    error = MACFile_Seek(hFile, (unsigned long)value, mode)

#define IL_SEEK_BEGIN        SEEK_SET
#define IL_SEEK_CURRENT      SEEK_CUR
#define IL_SEEK_END          SEEK_END

```



```

#define IL_TELL(hFile, val)    val = MACFile_Tell (hFile)
#define IL_MEMCHR              memchr
#define IL_MEMCMP              memcmp
#define IL_MEMCPY              memcpy
#define IL_MEMMOVE             memmove
#define IL_MEMSET              memset
#define IL_STRCAT              strcat
#define IL_STRCHR              strchr
#define IL_STRCMP(x,lx,y,ly)   strcmp(x,y)
#define IL_STRCOMP             strcmp
#define IL_STRICMP             ILUT_stricmp
#define IL_STRCPY              strcpy
#define IL_STRCSPN             strcspn
#define IL_STRLEN              strlen
#define IL_STRNCAT             strncat
#define IL_STRNCMP             strncmp
#define IL_STRNCPY             strncpy
#define IL_STRNICMP            ILUT_strnicmp
#define IL_STRRCHR             strrchr
#define IL_STRSPN              strspn
#define IL_STRSTR              strstr
#define IL_STRTOK              strtok
#define IL_STRDUP(s,r)         {
                                IL_PSTR p;
                                p = (char *)malloc (IL_STRLEN (s)+1);
                                if (p) IL_STRCPY (p,s);
                                r = p;
                                }

//----- MAC TextUtils.h
#define IL_STRLWR(s)           LowerText (s, strlen(s))
#define IL_STRUPR(s)           UpperText (s, strlen(s))

#define IL_SPLITPATH           MACFile_SplitPath
#define IL_MAKEPATH            MACFile_MakePath
#define IL_GETTEMPFILENAME     MACFile_TempName

#define IL_CHSIZE(file, len)   MACFile_SetEOF (file, len)
#define IL_DOES_EXIST(file)    (MACFile_Exists (file))
#define IL_EXISTS(file, flag)  flag = MACFile_Exists (file)
#define IL_RENAME(n1, n2, rc)  (rc = rename (n1, n2))
#define IL_BEEP()              printf ("\a")
#define IL_CHDIR(dir, rc)      (rc = MACFile_SetDir (dir))
#define IL_GETCURDIR(dir, len, rc) (rc = MACFile_GetDir (dir))
#define IL_GETDRIVE(drive)     (drive)[0] = '\0'
#define IL_GETDIR(drv, dir, len, rc) (rc = MACFile_GetDir (dir))
#define IL_GETPID(nPid)        nPid = 1234;  //***** NOT IMPLEMENTED!
#define IL_MKDIR(dir, rc)      (rc = MACFile_MakeDir (dir))
#define IL_RMDIR(dir, rc)      //***** NOT IMPLEMENTED!
#define IL_SETDIR(dir)         MACFile_SetDir (dir)
#define IL_SETDRIVE(drv)       //***** NO-OP on MAC

/*-----
 * The following macros call MAC-Specific code designed
 * to emulate some commonly used Windows functions.
 *-----*/
#define LoadDll(name, hDll)    \
    hDll = ILMAC_LoadLibrary (name)
#define UnloadDll(hDll)        \
    ( if (hDll != NULL) ILMAC_FreeLibrary (hDll); )

#define LoadString(inst, id, buf, len) \
    ILMAC_LoadString (inst, id, buf, len)
#define GetModuleFileName(inst, path, len) \
    (( MACFile_AppPath (path) == 0) ? 0 : IL_STRLEN(path))

#define GetPrivateProfileString(sec, var, def, val, len, file) \
    IL_GetPrivateProfileString(sec, var, def, val, len, file)
#define WritePrivateProfileString(sec, var, val, file) \
    IL_WritePrivateProfileString(sec, var, val, file)

#define GetProfileString(sec, var, def, val, len) \
    IL_GetProfileString(sec, var, def, val, len)
#define WriteProfileString(sec, var, val) \
    IL_WriteProfileString(sec, var, val)

```

```

#define GetPrivateProfileInt(sec, var, def, file) \
    IL_GetPrivateProfileInt(sec, var, def, file)
#define GetProfileInt(sec, var, def) \
    IL_GetProfileInt(sec, var, def)

/*-----
 * The Mac concept of locking and unlocking memory is different from
 * Windows. Relocatable memory is either locked or unlocked. There is no
 * concept of a lock count. Therefore, once a block of memory is locked,
 * it will remain locked until freed. GlobalUnlock does nothing on the MAC.
 *-----*/
#define GlobalAlloc(f,n)      MacMem_Alloc ((UINT32) n)
#define GlobalReAlloc(h,n,f)  MacMem_Realloc ((UINT32) n, h)
#define GlobalFree(h)         MacMem_Free (h)

#define GlobalLock(h)         MacMem_Lock (h)
#define GlobalUnlock(a)       0

/*-----
 * Other Windows functions not available on the Mac
 *-----*/
#define IsCharAlpha(c)        isalpha (c)
#define IsCharAlphaNumeric(c) isalnum (c)
#define AnsiLower(s)          LowerText (s, strlen(s))
#define AnsiUpper(s)          UpperText (s, strlen(s))

//----- File pattern matching/enumeration - NOT IMPLEMENTED!!!!
#define IL_FINDCLOSE(h,rc)
#define IL_FINDFIRST(nm,at,st,h,rc)
#define IL_FINDNEXT(h,st,rc)

//----- do pointer subtraction and get INT32 result. This is trivial
//----- because PTRDIFF_T is long for this environment.
#define IL_PTRDIFF(a,b) (b-a)
#define IL_HUGE_PTRDIFF(a,b) (b-a)

/*-----
 * Use IL_HANDLE_PADDING after any IL_HANDLE member of any structure that
 * needs to be size-invariant from one platform to another. For each plat-
 * form sizeof(IL_HANDLE) + sizeof(IL_HANDLE_PADDING) = 4 bytes.
 *-----*/
#define IL_HANDLE_PADDING(_name) // no padding needed for MAC

#endif

/*-----
 * The following macros are system-independent.
 *-----*/
#define IL_DOESNT_EXIST(filename) (!IL_DOES_EXIST(filename))

/*-----
 * Use the next 6 macros to improve readability of code that does
 * some common low-level operations for null-terminated strings.
 *-----*/
#define IL_STRINGS_EQUAL(a,b)      (IL_STRCOMP(a,b)==0)
#define IL_STRINGS_EQUALN(a,b,len) (IL_STRNCMP(a,b,len)==0)
#define IL_STRINGS_CI_EQUAL(a,b)   (IL_STRICMP(a,b)==0)

#define IL_MAKE_STRING_NULL(s)      (s)[0]=0
#define IL_STRING_IS_NULL(s)        ((s)[0] == 0)
#define IL_STRING_ISNT_NULL(s)      ((s)[0] != 0)

/*-----
 * Use the next 2 macros to copy a string into a buffer while
 * guarding against 2 hazards:
 * 1. overflowing the target buffer,
 * 2. failing to null-terminate the result
 *-----*/

#define IL_SAFE_STRINGCOPY(target,source) \
{ \
    IL_STRNCPY(target, source, sizeof(target)-1); \
    (target) [sizeof(target)-1] = 0; \
}

```

```

#define IL_SAFE_STRINGCOPYN(target, source, n) \
{ \
    if (n < 2) \
        (target)[0] = 0; \
    else \
    { \
        target[0] = '\0'; \
        IL_STRNCAT(target, source, n-1); \
    } \
}

/*-----
 * IL_SAFE_STRINGS_EQUAL builds anti-GPF protection into vanilla
 * string compare to check for equality.
 *-----*/
#define IL_SAFE_STRINGS_EQUAL(p,q) \
    (((p)==(q)) ? TRUE : \
    (((p)==NULL || (q)==NULL) ? FALSE : \
    (IL_STRCOMP((p),(q))==0) \
    ))

/*-----
 * IL_SAFE_STRINGS_NOT_EQUAL builds anti-GPF protection into vanilla
 * string compare to check for inequality.
 *-----*/
#define IL_SAFE_STRINGS_NOT_EQUAL(p,q) \
    (((p)==(q)) ? FALSE : \
    (((p)==NULL || (q)==NULL) ? TRUE : \
    (IL_STRCOMP((p),(q))!=0) \
    ))

/*-----
 * Use the ILERROR macro to log serious unexpected errors.
 *
 * both args are of type 'int'.
 *
 * Typical usages are as follows:
 *
 * Example #1:
 *
 *     rc = f(...)
 *     if (rc != SUCCESS)
 *         return ILERROR(rc, ILTR_ERR_INTERNAL_ERROR);
 *
 * Example #2:
 *
 *     rc = f(...)
 *     if (rc != SUCCESS)
 *         EXIT_WITH_ERROR (ILERROR(rc, ILTR_ERR_INTERNAL_ERROR));
 *
 * Example #3:
 *
 *     rc = f(...)
 *     if (rc != SUCCESS)
 *         ILERROR(rc, 0);
 *
 *     //..... keep on processing after logging error .....
 *
 * Use ILERROR when underlyingEC is an int.
 * Use ILERROR_L when underlyingEC is an INT32 (long).
 * Use ILERROR_S when underlyingEC is a string.
 *
 *-----*/
#define ILERROR(underlyingEC, ECtoReturn) \
    ILUT_Error ( __FILE__, __LINE__, underlyingEC, ECtoReturn)

#define ILERROR_L(underlyingEC, ECtoReturn) \
    ILUT_Error_L ( __FILE__, __LINE__, underlyingEC, ECtoReturn)

#define ILERROR_S(underlyingEC, ECtoReturn) \
    ILUT_Error_S ( __FILE__, __LINE__, underlyingEC, ECtoReturn)

/*-----
 * The following ILTRACE macros are useful for writing non-error

```

```

* log lines to the ILERRORS.LOG file.
*
*      ILTRACE0 adds a high-precision timestamp to the text that you
*      supply
*
*      ILTRACE1 adds a high-precision timestamp + file & line#
*      to the text that you supply
*-----*/
#define ILTRACE0(szText) ILUT_Trace0 (szText)

#define ILTRACE1(szText) ILUT_Trace1 (__FILE__, __LINE__, szText)

// ----- if min and max have not yet been defined, define them now

#ifndef NOMINMAX          // use NOMINMAX to turn off min/max macros
#ifndef max
#define max(a,b)          (((a) > (b)) ? (a) : (b))
#endif
#ifndef min
#define min(a,b)          (((a) < (b)) ? (a) : (b))
#endif
#endif // NOMINMAX

//----- General macros (adapted from Microsoft Windows versions)

#ifndef LOBYTE
#define LOBYTE(w)         ((BYTE)(w))
#endif
#ifndef HIBYTE
#define HIBYTE(w)         ((BYTE)(((UINT16)(w) >> 8) & 0xFF))
#endif
#ifndef LOWORD
#define LOWORD(l)          ((WORD)(DWORD)(l))
#endif
#ifndef HIWORD
#define HIWORD(l)          ((WORD)(((DWORD)(l) >> 16) & 0xFFFF))
#endif
#ifndef MAKELONG
#define MAKELONG(low, high)((LONG)(((WORD)(low))|(((DWORD)((WORD)(high)))<<16)))
#endif
#ifndef MAKEUINT
#define MAKEUINT(low, high) ((UINT)(((BYTE)(low))|(((UINT)((BYTE)(high)))<<8)))
#endif
//----- following defines are to allow Mac compilation
#ifndef IDOK
#define IDOK               1
#endif
#ifndef IDCANCEL
#define IDCANCEL           2
#endif

//-----
// Macro to "ones complement" a null terminated character string.
// Note that "a" must be char[n], not IL_PSTR, since this macro
// uses sizeof as part of the loop test to catch the case of
// improperly formatted file (no trailing null)
//-----
#define IL_ONES_COMP(a) \
{ \
    unsigned int i; \
    for (i = 0; i < min (sizeof(a), IL_STRLEN(a)); i++) \
        a[i] = ~a[i]; \
}

//-----
// The tr structure and structures contained within the tr structure
// must be padded to make the 16 bit structure and the 32 bit structure
// come out the same size. This is done by using the following macro
// in the places in those structures where the size varies. You give
// it the name of the field as the first parameter and how many bytes
// need to be padded as the second(This is the WIN32 size minus the
// WIN16 size. Notice that this macro expands to a no op in the case
// where ILX32_16 is not defined. This is to maintain backwards
// compatibility with our 3.2.2 version and before. The only place
// this should be defined is when compiling a 16 bit version that will

```

```
// need to except a tr structure from a win32 app.
//-----

#ifndef ILX32_16
#define ILX32_16PAD(name, size)
#else
#define ILX32_16PAD(name, size)    char name[size];
#endif    // ILX32_16

//----- Several of the above MAC macros require ILUTIL, so we include ILUTIL.H.
#ifdef ILMAC
    #include "ilutil.h"
#endif

//----- Several of the above WIN macros require ILUTIL, so we include ILUTIL.H.
#ifdef ILWIN
    #include "ilutil.h"
#endif

#endif // __ILMACRO
```

```

#if !defined(__ILTIME)

/*-----
* Name:      ILTIME.H
* Purpose:   Header file for IntelliLink date and time routines
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992-1995
* Notes:
*   This header file defines symbols and function prototypes for the
*   IntelliLink time library. These functions are used to convert
*   between three distinct date/time formats: ALPHA, DISPLAY, and
*   ENCODED. When stored in the intermediate file, all date and time
*   fields are stored using the ALPHA format. When a field is shown
*   to the user, it is always converted to the DISPLAY format. For
*   internal computations, the date and time fields are generally stored
*   in the ENCODED format.
*
*   The format of ALPHA date fields is YYYYMMDD (ex. 19941031). The
*   format of DISPLAY date fields is controlled by format parameters
*   held in an ILTM_DTTM_FMT structure. (discussed further below)
*
*   Usually the DISPLAY format is set to follow the natural format rules of
*   the underlying system. Under Windows, this is taken directly from
*   the user definition supplied for Short Date Format (changeable from
*   Control Panel). The ENCODED date format is the number of days
*   spanned since January 1, 1900.
*
*   The format of ALPHA time fields is HHMM (ex. 1430). The format of
*   DISPLAY time fields is controlled by format parameters held in an
*   ILTM_DTTM_FMT structure. (discussed further below)
*
*   Usually the DISPLAY format is set to follow the rules of the underlying
*   system.
*
*   Under Windows, this is taken from the time format specified under
*   International in the Control Panel. Both other systems (like DOS),
*   the format is defaulted to an applicable display format. The
*   ENCODED time format is computed as the number of seconds elapsed
*   since midnight.
*
* Use of ILTM_DTTM_FMT parameter blocks:
*-----
*
*   All of the date/time functions that build or parse DISPLAY formats
*   use parameters held in an ILTM_DTTM_FMT parameter block. The "standard"
*   format parameters are kept in a GLOBAL VARIABLE parameter block
*   called ILTM_StdFmt.
*
*   Sometimes one needs to use non-standard format parameters. Several
*   translators have this requirement. To accommodate them, a secondary
*   parameter block is kept in the "tr" structure -- ILTR_DtTmFmt.
*
*   You are also free to maintain your own ILTM_DTTM_FMT parameter block,
*   e.g. if you don't pass the "tr" pointer around you might want to hang
*   a parameter block off your "gl" pointer.
*
*   Regardless of what parameter block you use, you must ensure that it is
*   initialized before you use it. Initialization is ensured for all
*   translators by the following lines in ILTR\EXPORT.C and IMPORT.C:
*
*       //----- Set ILTIME global default date and time format
*       IL_InitStdDateFormat ();
*       IL_InitStdTimeFormat ();
*
*       //----- set ILTR date and time format
*       IL_InitDateFormat ();
*       IL_InitTimeFormat ();
*
*   Note that BOTH of the "built in" parameter blocks are initialized.
*   There are many translators which always use the "standard" DISPLAY
*   formats. Given the identical initializations of the two parameter
*   blocks, such translators are free to use either one!! For historical
*   reasons there are many examples of each usage pattern.
*
* Base-level functions and convenience macros for DISPLAY-related functions
*-----

```



```

*
*   There are 12 base-level functions that deal with DISPLAY formats in
*   some way:
*
*       4 "Set/Tell" functions:      IL_<Set/Tell><Date/Time>FormatEx
*       4 "DisplayToXXX" functions:  IL_DisplayTo<Code/><Date/Time>Ex
*       4 "XXXToDisplay" functions:  IL_<Code/><Date/Time>ToDisplayEx
*
*   Each of these base-level functions has TWO associated macros. For
*   example, here is a typical function and its two macros:
*
*       IL_DisplayToDateEx      -- base-level function
*       IL_DisplayToDate        -- macro to call with ILTR_DtTmFmt
*       IL_StdDisplayToDate      -- macro to call with ILTM_StdFmt
*
*   In addition to these macros there are 4 "Init" macros which are
*   used to call the "Set Format" base-level functions.
*
*   Please use the convenience macros!! Avoid calling the base-level
*   functions directly whenever possible. You should only call the
*   base-level functions if you need to supply your own private
*   ILTM_DTTM_FMT parameter block.
*
*-----*/
#define __ILTIME                                // Signal header inclusion
#ifdef __cplusplus                               // Special handling for C++
    #pragma warning (disable: 4200)
    extern "C" {
#endif // __cplusplus

/*-----
* Header files.
*-----*/
#ifdef ILWIN                                     // Windows headers
    #if !defined(__WINDOWS)
        #include <windows.h>
        #define __WINDOWS
    #endif
#endif
#include <stdlib.h>
#include <string.h>
#include "ilutil.h"

/*-----
* Symbolic constants.
*-----*/
#ifndef SUCCESS
    #define SUCCESS          0                // No error
#endif
#ifndef FAILURE
    #define FAILURE          -1               // Error found
#endif
#define IL_12_HOURS          0                // 12 hour time format
#define IL_24_HOURS          1                // 24 hour time format
#define IL_NO_LEAD           0                // No leading character in time
#define IL_LEAD_ZERO         1                // Leading zero in time field
#define IL_LEAD_BLANK        2                // Leading blank in time field
#define IL_NONE              "\0"            // No value supplied
#define ILTM_NOTSET          '\xFF'          // param hasn't been initialized

/*-----
* Limits.
*-----*/
#define ILTM_MAX_SUFFIX      4                // Max size of time suffix string
#define ILTM_MAX_DATE_SPEC   11              // Max size of date format

/*-----
* Error codes.
*-----*/
#define ILTM_ERR_SPEC        6000             // Invalid date or time format
#define ILTM_ERR_MINS        6001             // Invalid minutes
#define ILTM_ERR_HOURS       6002             // Invalid hours
#define ILTM_ERR_SECS        6003             // Invalid seconds

```

```

#define ILTM_ERR_MONTH      6004          // Invalid month
#define ILTM_ERR_DAY        6005          // Invalid day
#define ILTM_ERR_YEAR       6006          // Invalid year
#define ILTM_ERR_BADTIME    6007          // Invalid time string
#define ILTM_ERR_BADDATE    6008          // Invalid date string

/*-----
 * Date&Time Format Parameter Blocks
 *-----*/
typedef struct
{
    char cTimeType;    // 12 or 24 hour time format
    char cTimeLead;    // Lead character in time field (zero, blank, or none)
    char cTimeSep;     // Time separator
    char cDateSep;     // Date separator
    char szAM[ILTM_MAX_SUFFIX]; // AM string
    char szPM[ILTM_MAX_SUFFIX]; // PM string
    char szDateFormat[ILTM_MAX_DATE_SPEC];
}
ILTM_DTTM_FMT;

typedef ILTM_DTTM_FMT IL_DIST *ILTM_DTTM_FMT_PTR;

extern ILTM_DTTM_FMT ILTM_StdFmt;

/*-----
 * Initialization macros for the ILTIME GLOBAL Standard Format Parameters
 *-----*/
#define IL_InitStdDateFormat() \
    ( if (ILTM_StdFmt.cDateSep == ILTM_NOTSET) \
      IL_SetStdDateFormat ("\0", '\0'); )

#define IL_InitStdTimeFormat() \
    ( if (ILTM_StdFmt.cTimeType == ILTM_NOTSET) \
      IL_SetStdTimeFormat (-1, '\0', 0, "\0", "\0" ); )

/*-----
 * Initialization macros for the ILTR Format Parameters
 *-----*/
#define IL_InitDateFormat() IL_SetDateFormat ("\0", '\0')
#define IL_InitTimeFormat() IL_SetTimeFormat (-1, '\0', 0, "\0", "\0" )

/*-----
 * Convenience macros for working with STANDARD DISPLAY format.
 *-----*/
#define IL_CodeDateToStdDisplay(_lDate, _szDate) \
    IL_CodeDateToDisplayEx (&ILTM_StdFmt, _lDate, _szDate)

#define IL_CodeTimeToStdDisplay(_lTime, _szTime) \
    IL_CodeTimeToDisplayEx (&ILTM_StdFmt, _lTime, _szTime)

#define IL_DateToStdDisplay(_nMonth, _nDay, _nYear, _szDate) \
    IL_DateToDisplayEx (&ILTM_StdFmt, _nMonth, _nDay, _nYear, _szDate)

#define IL_StdDisplayToCodeDate(_szDate, _plDate) \
    IL_DisplayToCodeDateEx (&ILTM_StdFmt, _szDate, _plDate)

#define IL_StdDisplayToCodeTime(_szTime, _plTime) \
    IL_DisplayToCodeTimeEx (&ILTM_StdFmt, _szTime, _plTime)

#define IL_StdDisplayToDate(_szDate, _pnMonth, _pnDay, _pnYear) \
    IL_DisplayToDateEx (&ILTM_StdFmt, _szDate, _pnMonth, _pnDay, _pnYear)

#define IL_StdDisplayToTime(_szTime, _pnHours, _pnMins) \
    IL_DisplayToTimeEx (&ILTM_StdFmt, _szTime, _pnHours, _pnMins)

#define IL_SetStdDateFormat(_szFormat, _cSep) \
    IL_SetDateFormatEx (&ILTM_StdFmt, _szFormat, _cSep)

#define IL_SetStdTimeFormat(_nType, _cSep, _nLead, _szAM, _szPM) \
    IL_SetTimeFormatEx (&ILTM_StdFmt, _nType, _cSep, \
        _nLead, _szAM, _szPM )

#define IL_TellStdDateFormat(_szFormat, _pcSep) \
    IL_TellDateFormatEx (&ILTM_StdFmt, _szFormat, _pcSep)

```

```

#define IL_TellStdTimeFormat(_pnType, _pcSep, _pnLead, _szAM, _szPM) \
    IL_TellTimeFormatEx ( &ILTM_StdFmt, _pnType, _pcSep, \
        _pnLead, _szAM, _szPM )

#define IL_TimeToStdDisplay(_nHours, _nMins, _szTime) \
    IL_TimeToDisplayEx (&ILTM_StdFmt, _nHours, _nMins, _szTime)

/*-----
 * Convenience macros for working with current ILTR DISPLAY format.
 *-----*/
#define IL_CodeDateToDisplay(_lDate, _szDate) \
    IL_CodeDateToDisplayEx (&ILTR_DtTmFmt, _lDate, _szDate)

#define IL_CodeTimeToDisplay(_lTime, _szTime) \
    IL_CodeTimeToDisplayEx (&ILTR_DtTmFmt, _lTime, _szTime)

#define IL_DateToDisplay(_nMonth, _nDay, _nYear, _szDate) \
    IL_DateToDisplayEx (&ILTR_DtTmFmt, _nMonth, _nDay, _nYear, _szDate)

#define IL_DisplayToCodeDate(_szDate, _plDate) \
    IL_DisplayToCodeDateEx (&ILTR_DtTmFmt, _szDate, _plDate)

#define IL_DisplayToCodeTime(_szTime, _plTime) \
    IL_DisplayToCodeTimeEx (&ILTR_DtTmFmt, _szTime, _plTime)

#define IL_DisplayToDate(_szDate, _pnMonth, _pnDay, _pnYear) \
    IL_DisplayToDateEx (&ILTR_DtTmFmt, _szDate, _pnMonth, _pnDay, _pnYear)

#define IL_DisplayToTime(_szTime, _pnHours, _pnMins) \
    IL_DisplayToTimeEx (&ILTR_DtTmFmt, _szTime, _pnHours, _pnMins)

#define IL_SetDateFormat(_szFormat, _cSep) \
    IL_SetDateFormatEx (&ILTR_DtTmFmt, _szFormat, _cSep)

#define IL_SetTimeFormat(_nType, _cSep, _nLead, _szAM, _szPM) \
    IL_SetTimeFormatEx ( &ILTR_DtTmFmt, _nType, _cSep, \
        _nLead, _szAM, _szPM )

#define IL_TellDateFormat(_szFormat, _pcSep) \
    IL_TellDateFormatEx (&ILTR_DtTmFmt, _szFormat, _pcSep)

#define IL_TellTimeFormat(_pnType, _pcSep, _pnLead, _szAM, _szPM) \
    IL_TellTimeFormatEx ( &ILTR_DtTmFmt, _pnType, _pcSep, \
        _pnLead, _szAM, _szPM )

#define IL_TimeToDisplay(_nHours, _nMins, _szTime) \
    IL_TimeToDisplayEx (&ILTR_DtTmFmt, _nHours, _nMins, _szTime)

/*-----
 * Function prototypes.
 *-----*/
BOOLEAN IL_DECL IL_AlphaDateOK          // Is date string valid?
( IL_PSTR sDate );                     // Date string
BOOLEAN IL_DECL IL_AlphaTimeOK          // Is time string valid?
( IL_PSTR sTime );                     // Time string
long IL_DECL IL_AlphaToCodeDate         // convert to days since 1900
( IL_PSTR date );                      // IN: e.g. "19951231"
long IL_DECL IL_AlphaToCodeTime         // convert to seconds since 12AM
( IL_PSTR time );                      // IN: e.g. "2359"
void IL_DECL IL_AlphaToDate             // Split date string
( IL_PSTR date,                        // Pointer to date string
    int IL_DIST *month,                // Pointer to month number
    int IL_DIST *day,                  // Pointer to day number
    int IL_DIST *year );               // Pointer to year number
void IL_DECL IL_AlphaToTime             // Split time string
( IL_PSTR time,                        // Pointer to time string
    int IL_DIST *hours,                // Pointer to hours
    int IL_DIST *mins );               // Pointer to minutes
IL_PSTR IL_DECL IL_CodeDateToAlpha      // Convert encoded date to YYYYMMDD
( long lDate,                          // IN: days since 1900
    IL_PSTR pszDate );                 // OUT: e.g. "19951231"
IL_PSTR IL_DECL IL_CodeTimeToAlpha      // Convert encoded time to HHMM
( long lTime,                          // IN: seconds since midnight

```

```

        IL_PSTR pszTime );
IL_PSTR IL_DECL IL_CodeDateToDisplayEx
( ILTM_DTTM_FMT_PTR pFmt,
  long lDate,
  IL_PSTR pszDate );
IL_PSTR IL_DECL IL_CodeTimeToDisplayEx
( ILTM_DTTM_FMT_PTR pFmt,
  long lTime,
  IL_PSTR pszTime );
int IL_DECL IL_DateDecode
( long date,
  int IL_DIST *month,
  int IL_DIST *day,
  int IL_DIST *year );
int IL_DECL IL_DateEncode
( int month,
  int day,
  int year,
  long IL_DIST *date );
IL_PSTR IL_DECL IL_DateToAlpha
( int month,
  int day,
  int year,
  IL_PSTR date );
IL_PSTR IL_DECL IL_DateToDisplayEx
( ILTM_DTTM_FMT_PTR pFmt,
  int month,
  int day,
  int year,
  IL_PSTR date );
int IL_DECL IL_DaysInMonth
( int month,
  int year );
int IL_DECL IL_DayOfWeek
( long date );
int IL_DECL IL_DisplayToCodeDateEx
( ILTM_DTTM_FMT_PTR pFmt,
  IL_PSTR pszDate,
  long IL_DIST *plDate );
int IL_DECL IL_DisplayToCodeTimeEx
( ILTM_DTTM_FMT_PTR pFmt,
  IL_PSTR pszTime,
  long IL_DIST *plTime );
int IL_DECL IL_DisplayToDateEx
( ILTM_DTTM_FMT_PTR pFmt,
  IL_PSTR date,
  int IL_DIST *month,
  int IL_DIST *day,
  int IL_DIST *year );
int IL_DECL IL_DisplayToTimeEx
( ILTM_DTTM_FMT_PTR pFmt,
  IL_PSTR time,
  int IL_DIST *hours,
  int IL_DIST *mins );
long IL_DECL IL_GetCurrentDate ();
long IL_DECL IL_GetCurrentTime ();
int IL_DECL IL_SetDateFormatEx
( ILTM_DTTM_FMT_PTR pFmt,
  IL_PSTR format,
  char sep );
int IL_DECL IL_SetTimeFormatEx
( ILTM_DTTM_FMT_PTR pFmt,
  int type,
  char sep,
  int lead,
  IL_PSTR am,
  IL_PSTR pm );
int IL_DECL IL_TellDateFormatEx
( ILTM_DTTM_FMT_PTR pFmt,
  IL_PSTR format,
  char IL_DIST *sep );
int IL_DECL IL_TellTimeFormatEx
( ILTM_DTTM_FMT_PTR pFmt,
  int IL_DIST *type,
  char IL_DIST *sep,
  // OUT: e.g. "2359"
  // Convert encoded date to display format
  // Pointer to format params block
  // Encoded date value
  // Pointer to date string
  // Convert encoded time to display format
  // Pointer to format params block
  // Encoded time value
  // Pointer to time string
  // Decode date field
  // Encoded date value
  // Pointer to month number
  // Pointer to day number
  // Pointer to year number
  // Encode date field
  // Month number (ex. 10)
  // Day number (ex. 31)
  // Year number (ex. 1994)
  // Pointer to date string
  // Construct date string
  // Month number (ex. 10)
  // Day number (ex. 31)
  // Year number (ex. 1994)
  // Pointer to date string
  // Convert to display format
  // Pointer to format params block
  // Month number (ex. 10)
  // Day number (ex. 31)
  // Year number (ex. 1994)
  // Pointer to date string
  // How many days in given month?
  // Month number (ex. 10)
  // Year number (ex. 1994)
  // What is day of week for date?
  // Encoded date value
  // Convert display format to encoded date
  // Pointer to format params block
  // Pointer to date string
  // Encoded date value
  // Convert display format to encoded time
  // Pointer to format params block
  // Pointer to time string
  // Encoded time value
  // Split display format date
  // Pointer to format params block
  // Pointer to date string
  // Pointer to month number
  // Pointer to day number
  // Pointer to year number
  // Split display format time
  // Pointer to format params block
  // Pointer to time string
  // Pointer to hours
  // Pointer to minutes
  // Get current encoded date
  // Get current encoded time
  // Set format of date strings
  // Pointer to format params block
  // Pointer to format string
  // Component separator
  // Set format of time strings
  // Pointer to format params block
  // Time format type
  // Component separator
  // Leading blanks or zeros
  // Pointer to AM string suffix
  // Pointer to PM string suffix
  // Tell current date format
  // Pointer to format params block
  // Pointer to format string
  // Pointer to date separator
  // Tell current time format
  // Pointer to format params block
  // Pointer to time format type
  // Pointer to time separator
);

```

```

        int IL_DIST *lead,
        IL_PSTR am,
        IL_PSTR pm );
int IL_DECL IL_TimeDecode
( long time,
  int IL_DIST *hours,
  int IL_DIST *mins,
  int IL_DIST *secs );
int IL_DECL IL_TimeEncode
( int hours,
  int mins,
  int secs,
  long IL_DIST *time );
IL_PSTR IL_DECL IL_TimeToAlpha
( int hours,
  int mins,
  IL_PSTR time );
IL_PSTR IL_DECL IL_TimeToDisplayEx
( ILTM_DTTM_FMT_PTR pFmt,
  int hours,
  int mins,
  IL_PSTR time );
int IL_DECL IL_WeekInMonth
( long date );

// Pointer to lead time type
// Pointer to AM string suffix
// Pointer to PM string suffix
// Decode time field
// Encoded time value
// Pointer to hours
// Pointer to minutes
// Pointer to seconds
// Encode time value
// Hours number (ex. 1)
// Minutes number (ex. 15)
// Seconds number (ex. 30)
// Encoded time field
// Create time string
// Hours number (ex. 1)
// Minutes number (ex. 15)
// Pointer to time string
// Create display format time
// Pointer to format params block
// Hours number (ex. 1)
// Minutes number (ex. 15)
// Pointer to time string
// Get week of week for date
// Encoded date field

#ifdef __cplusplus
}
#endif // __cplusplus

#endif // __ILTIME

```

```

// Special handling for C++

```

```

#ifndef __ILUTIL
/*-----
 * Name:      ILUTIL.H
 * Purpose: Header file for IntelliLink general purpose utility routines
 * Author: Copyright (c) IntelliLink, 1994
 *-----*/
#define __ILUTIL                // Signal header inclusion

#ifdef __cplusplus                // Special handling for C++
    #pragma warning (disable: 4200)
    extern "C" {
#endif // __cplusplus

/*-----
 * Header files.
 *-----*/
#ifdef ILWIN                // Windows headers
    #if !defined(__WINDOWS)
        #include <windows.h>
        #define __WINDOWS
    #endif // __WINDOWS
#endif // ILWIN

#include <stdlib.h>
#include <string.h>

#ifdef SYSMGR                // System Manager headers
    #include "interfac.h"
    #include "fileio.h"
    // Declare these here since we cannot include stdio.h
    int __cdecl sprintf(char *, const char *, ...);
    char * __cdecl tmpnam(char *);
#else
#ifdef RC_INVOKED
    #include <stdio.h>
    #ifndef ILMAC
        #include <io.h>
        #include <dos.h>
    #endif
#endif
#endif
#endif // SYSMGR

//----- Common Intellilink types, other utilities, et al
#include "ilio.h"

/*-----
 * Basic data types.
 *-----*/
#ifdef ILWIN                // Windows definitions
    #define IL_MAX_RESOURCE_LEN 14

    typedef struct
    {
        char name[IL_MAX_RESOURCE_LEN];    // Resource item name
    } IL_RCTYPE;

    typedef struct
    {
        IL_RCTYPE names[];                // Section type list
    } IL_RESOURCE;
#endif // ILWIN

// Different types of file validation
typedef enum
{
    ILUT_VALID_PARSE,                // Check file syntax only
    ILUT_VALID_DIRS,                // Check drive/directory existence
    ILUT_VALID_FILE,                // Check if file exists
    ILUT_VALID_UPDATE                // Check if file can be updated
} ILUT_VALID;

//----- Character codes
#define ILUT_BSLASH_CHAR    '\\\\'    // BACKSLASH
#define ILUT_BSLASH_STR    "\\\"    // BACKSLASH string

```

```

#define ILUT_COLON_CHAR      ':'      // COLON
#define ILUT_COLON_STR      ":"      // COLON string
#define ILUT_CR_CHAR        '\r'     // RETURN
#define ILUT_CR_STR         "\r"     // RETURN string
#define ILUT_LF_CHAR        '\n'     // NEWLINE
#define ILUT_LF_STR         "\n"     // NEWLINE
#define ILUT_CRLF_STR       ILUT_CR_STR ILUT_LF_STR // CRLF pair
#define ILUT_NULL_CHAR      '\0'     // NULL
#define ILUT_NULL_STR       "\0"     // NULL string
#define ILUT_PERIOD_CHAR    '.'      // PERIOD
#define ILUT_PERIOD_STR     "."      // PERIOD string
#define ILUT_SPACE_CHAR     ' '      // SPACE
#define ILUT_SPACE_STR      " "      // SPACE string
#define ILUT_TAB_CHAR       '\t'     // TAB
#define ILUT_TAB_STR        "\t"     // TAB string

#ifdef ILWIN
    #define ILUT_BUFSIZE      16384   // Windows buffer for file copy
#else
    #define ILUT_BUFSIZE      512     // DOS buffer for file copy
#endif // ILWIN

/*-----
 * define ILUT_MEMPTR, used by the MEMUTIL functions
 *-----*/
#ifdef STRICT
    #define ILUT_MEMPTR IL_PANY
#else
    #define ILUT_MEMPTR IL_PSTR
#endif

/*-----
 * Error codes.
 *-----*/
#define ILUT_ERR_NOFILE      7000     // File not found
#define ILUT_ERR_CREATE     7001     // Can't create file
#define ILUT_ERR_READ       7002     // Can't read file
#define ILUT_ERR_WRITE      7003     // Can't write file
#define ILUT_ERR_NOMEM      7004     // Can't allocate memory
#define ILUT_ERR_CHSIZE     7005     // chsize() failed
#define ILUT_ERR_BUFFER     7006     // Buffer too small
#define ILUT_ERR_DATA       7007     // Invalid data
#define ILUT_ERR_SIZE       7008     // Invalid size
#define ILUT_ERR_PARSE      7009     // Parse error
#define ILUT_ERR_DIR        7010     // Directory does not exist
#define ILUT_ERR_WRONGTYPE  7011     // ILFldGetLine got wrong line type
#define ILUT_ERR_SETTLE_FOR_LESS 7012 // you asked for more than max;
                                         // we're just giving you the max

/*-----
 * Function prototypes.
 *-----*/

int IL_DECL ILAsciiToHex          // Convert ASCII to HEX string
( IL_PSTR szSource,              // ASCII string buffer
  int nSrcLen,                   // Length of ASCII string
  IL_PSTR szTarget,              // Buffer to hold HEX string
  int nTarLen);                  // Size of HEX buffer

//----- Place holders, not yet written
int IL_chdir (IL_PSTR);          // Change current directory
int IL_chdrive (int);            // Change current drive
int IL_getcwd (int, IL_PSTR, int); // Get current directory
int IL_getdrive (void);          // Get current drive

UINT16 IL_DECL ILUT_ByteSwap16   // Swap bytes if "backwards"
( UINT16 uiValue );              // Value to operate upon
UINT32 IL_DECL ILUT_ByteSwap32   // Swap bytes if "backwards"
( UINT32 uiValue );              // Value to operate upon
int IL_DECL ILFldGetLine          // Get next line from text file
( IL_HFILE IL_DIST *stream,      // Pointer to file handle
  IL_PSTR type,                  // Line prefix to locate
  IL_PSTR line,                  // Line buffer to return
  int len );                     // Size of line buffer
int IL_DECL ILHexToAscii          // Convert HEX string to ASCII
( IL_PSTR szSource,              // HEX string buffer

```



```

        int nSrcLen,                // Length of HEX string
        IL_PSTR szTarget,          // Buffer to hold ASCII string
        int nTarLen);              // Size of ASCII string
int IL_DECL ILUT_CopyFile          // Copy a file
( IL_PCSTR szSourceFName,          // From file name
  IL_PCSTR szTargetFName);         // To file name
int IL_DECL ILUT_FileExist         // Determine whether file exists
( IL_PSTR szFileName);             // File name
BOOLEAN IL_DECL ILUT_FileIsDevice // Is the file a device?
( IL_PCSTR pFileName );            // File path
int IL_DECL ILUT_GetFileAttributes // Get the attributes of a file
( IL_PCSTR szFileName,             // File name
  int IL_DIST *pnFileAttr);        // Destination for attributes

/*-----
 * Note the subtle differences between ILUT_GetTempFile and
 * ILUT_GetTempFileName.  Function ILUT_GetTempFile returns the path
 * of a temporary file in a default directory and always creates a
 * placeholder for the returned file name.  ILUT_GetTempFileName
 * affords a greater degree of control by enabling the caller to
 * optionally specify a directory to contain the temporary file, a
 * prefix to use at the beginning of the file name, and a flag
 * to signal whether a placeholder should be created for the file.
 *-----*/
IL_PSTR IL_DECL ILUT_GetTempFile    // Create temporary file
( IL_PSTR lpPath );                 // Full path of temporary file
IL_PSTR IL_DECL ILUT_GetTempFileName // Get temporary file name
( IL_PSTR lpDir,                    // Location of temporary file
  IL_PSTR lpPrefix,                 // File name prefix to use
  IL_PSTR lpPath,                   // Full path returned
  BOOLEAN bCreate );                // Create placeholder file?
IL_PSTR IL_DECL ILUT_itoa           // Convert short to C-string
( int nValue,                       // Value to convert
  IL_PSTR pResult,                  // Result buffer pointer
  INT16 nRadix );                   // Base for conversion
BOOL16 ILUT_IsValidDirectory        // Is this a valid directory?
( IL_PCSTR pDir );                  // Pointer to directory path
BOOL16 ILUT_IsValidDrive            // Is this a valid drive?
( IL_PCSTR pPath );                 // Pointer to drive
BOOL16 ILUT_IsZeroFile              // Is this a zero-length file?
( IL_PSTR pFileName );              // Pointer to file name
int IL_DECL ILUT_LongToAscii        // Convert long to ASCII string
( long n,                           // Number to convert
  char IL_DIST *pszDst,              // Buffer to hold ASCII string
  int iMaxLen,                       // Size of buffer
  int iMinDigits);                  // Minimum digits to pad
IL_PSTR IL_DECL ILUT_ltoa           // Convert long to C-string
( INT32 lValue,                      // Value to convert
  IL_PSTR pResult,                   // Result buffer pointer
  INT16 nRadix );                   // Base for conversion

ILUT_MEMPTR IL_DECL ILUT_MemAlloc ( // Allocate memory, return ptr
  UINT32 nSize,                      // Size of allocation
  IL_HANDLE IL_DIST *phMem );        // Pointer to memory handle
void IL_DECL ILUT_MemFree (          // Free memory
  IL_HANDLE hMem,                    // Memory handle
  IL_PANY pMem );                   // Memory pointer
ILUT_MEMPTR IL_DECL ILUT_MemReAlloc ( // Reallocate memory, return ptr
  UINT32 nSize,                      // New size for allocation
  IL_HANDLE IL_DIST *phMem,          // Pointer to memory handle
  IL_PANY pMem );                   // old pointer (to be loggable)
void IL_DECL ILUT_MemSig (           // Set debug logging signature
  IL_PSTR pSig );                   // Pointer to memory handle

int IL_DECL ILUT_RenameFile          // Rename a file
( IL_PSTR szOldName,                 // Old file name
  IL_PSTR szNewName);               // New file name
int IL_DECL ILUT_SetFileSize         // Set the file size
( IL_PSTR szFname,                   // File name
  long newSize );                   // New file size
int ILUT_stricmp                     // Case insensitive string compare
( const char *src,                   // Source string
  const char *tar );                // Target string
int ILUT_strnicmp                    // Case insensitive sub-string comp.
( const char *src,                   // Source string

```



```

        const char *tar,
        unsigned int nCount );
int ILUT_strsub
( IL_PSTR pStr,
  UINT16 nMax,
  IL_PCSTR pFind,
  IL_PCSTR pRepl );
IL_PSTR IL_DECL ILUT_ultoa
( UINT32 lValue,
  IL_PSTR pResult,
  INT16 nRadix );
int IL_DECL ILUT_ValidFile
( IL_PSTR pFileName,
  ILUT_VALID nMode );
INT16 IL_DECL IL_axtoi
( IL_PSTR string );
INT32 IL_DECL IL_axtol
( IL_PSTR string );

int IL_DECL ILUT_Error
( IL_PSTR szCaller,
  int nLineNumber,
  int nUnderlyingErrorCode,
  int nErrorCodeToReturn );

int IL_DECL ILUT_Error_L
( IL_PSTR szCaller,
  int nLineNumber,
  INT32 lUnderlyingErrorCode,
  int nErrorCodeToReturn );

int IL_DECL ILUT_Error_S
( IL_PSTR szCaller,
  int nLineNumber,
  IL_PSTR lpszUnderlyingErrorCode,
  int nErrorCodeToReturn );

int IL_DECL ILUT_ErrorLogDelete();

int IL_DECL ILUT_Trace0
( IL_PSTR szText );

int IL_DECL ILUT_Trace1
( IL_PSTR szCaller,
  int nLineNumber,
  IL_PSTR szText );

//----- Utility functions for passing data via "Settings"

int IL_DECL ILUT_CreateXlatorIniFile ( // Create full xlator INI file name
  IL_HINST hInst,                     // Instance handle
  IL_PSTR pszFileName,                // In: translator name (i.e. "ILXCDF"
  UINT32 nFileName);                 // Out: resulting filename buffer
                                     // Max length for filename

int IL_DECL ILUT_CreateSettingsKey ( // Create a new settings key
  IL_PSTR pszKeyName,                 // Key name (file name for 16-bit)
  IL_HKEY *phKey);                   // Pointer to IL_HKEY return value

int IL_DECL ILUT_FreeSettingsKey ( // Free a key
  IL_HKEY hKey );                     // IL_HKEY value

int IL_DECL ILUT_GetNumberSetting ( // Get a 4-byte int value
  IL_HKEY hKey,                       // Key for current section
  IL_PSTR pszSettingName,             // Keyword
  INT32 nDefault,                     // Default value if no setting found
  INT32 *pnSettingValue);             // Result value pointer

int IL_DECL ILUT_GetStringSetting ( // Get a string value
  IL_HKEY hKey,                       // Key for current section
  IL_PSTR pszSettingName,             // Keyword
  IL_PSTR pszDefault,                 // Default value if no setting found
  IL_PSTR pszSettingValue,            // Result string
  INT16 nResultSize);                // Max size for result

```

```

int IL_DECL ILUT_PutNumberSetting ( // Put a 4-byte int value
    IL_HKEY hKey, // Key for current section
    IL_PSTR pszSettingName, // Keyword
    INT32 nSettingValue); // New value to be set

int IL_DECL ILUT_PutStringSetting ( // Put a string value
    IL_HKEY hKey, // Key for current section
    IL_PSTR pszSettingName, // Keyword
    IL_PSTR pszSettingValue); // New value to be set

//----- The follow functions work on Windows only
#ifdef ILWIN

void IL_DECL ILUT_CenterWindow // Center window on parent
    ( IL_HWIN hParent, // Handle to parent or NULL
      IL_HWIN hChild, // Handle to child window
      BOOL16 bOnScreen ); // Force child on screen

HGLOBAL IL_DECL ILUT_GetDBNames // Get list of ODBC or other database names
    ( INT16 nSysID, // System ID
      LPCSTR pFileExt, // Ptr to database "file extension" string
      LPCSTR pFileName ); // Pointer to database name

int IL_DECL ILUT_ValidatedBName // Validate ODBC or other database name
    ( INT16 nSysID, // System ID
      LPCSTR pFileExt, // Ptr to database "file extension" string
      LPCSTR pFileName ); // Pointer to database name

HGLOBAL IL_DECL ILUT_OSWGetDBNames // Get list of On Schedule" database names
    ( INT16 nSysID, // System ID
      LPCSTR pFileExt, // Ptr to database "file extension" string
      LPCSTR pFileName ); // Pointer to database name

int IL_DECL ILGetResource // Load custom resource
    ( IL_HINST hInst, // Instance handle of owner
      IL_PSTR szRCName, // Resource name
      IL_PSTR szRType, // Resource type
      IL_HANDLE *hArray, // Pointer to memory handle
      IL_RESOURCE **pArray ); // Pointer to memory

void SetDialogBkColor // set dialog backgrnd/text color
    ( COLORREF clrCtlBk, // background color
      COLORREF clrCtlText ); // text color

void ResetDialogBkColor (); // reset SetDialogBkColor "hook"

void IL_DECL ILUT_TileWindow // Tile child window on parent
    ( IL_HWIN hParent, // Handle to parent or NULL
      IL_HWIN hChild ); // Handle to child window

BOOLEAN IL_DECL ILUT_TestCommPort // Test availability of specified COM port
    ( IL_PSTR szCommPort ); // Pointer to name of COM port to be tested

//----- Function used only with Lotus Organizer 2.1
HINSTANCE ILUT_LoadLotusAPI // Load ORGAPI and dependent DLLS
    ( LPCSTR lpInstallDir ); // IntelliLink directory
void ILUT_UnloadLotusAPI // Unload ORGAPI DLL
    ( HINSTANCE hOrgDLL ); // DLL instance handle

//----- Functions used only with Now Up-to-Date for Windows (3.1/95)
HINSTANCE ILUT_LoadNowAPI // Load NOWAPI and dependent DLLS
    ( LPCSTR lpInstallDir ); // IntelliLink directory
void ILUT_UnloadNowAPI // Unload NowAPI DLL
    ( HINSTANCE hNowDLL ); // DLL instance handle

//----- Function used only with Schedule+ 7.0
HINSTANCE ILUT_LoadSPlusAPI // Load Schedule+
    ( LPCSTR lpInstallDir ); // IntelliLink directory
void ILUT_UnloadSPlusAPI // Unload Schedule+
    ( HINSTANCE hSchDLL ); // DLL instance handle

/*-----
* Function: WinFile_Open
* Purpose: Generic Windows open routine, used by all IL functions.
* Returns: -1 if error, otherwise 0

```

```

/*-----*/
int IL_DECL WinFile_Open ( IL_PCSTR path,
                           int openMode,
                           LPOFSTRUCT pInfo,
                           IL_HFILE *phFile,
                           IL_PCSTR SourceFileCalledFrom,
                           int LineNumberCalledFrom );

/*-----*/
* Function: WinFile_Close
* Purpose:  Generic Windows close routine, used by all IL functions.
* Returns:  -1 if error, otherwise 0
/*-----*/
int IL_DECL WinFile_Close ( IL_HFILE hFile,
                           IL_PCSTR SourceFileCalledFrom,
                           int LineNumberCalledFrom );

#endif // ILWIN

#if defined (ILWIN) || defined (ILMAC)
//----- The following functions work on Windows and Macintosh only
long IL_DECL ILUT_GetFileSize      // Determine size of open file
    ( IL_HFILE hFile );           // File handle

#endif // ILWIN or ILMAC

#ifdef ILMAC
//----- The following functions work on Macintosh only
UINT16 IL_DECL ILUT_MacByteSwap16 // Swap bytes if "backwards" and on the mac
    ( UINT16 uiValue );           // Value to operate upon
UINT32 IL_DECL ILUT_MacByteSwap32 // Swap bytes if "backwards" and on the mac
    ( UINT32 uiValue );           // Value to operate upon
int IL_DECL ILMAC_LoadString       // Emulate Windows LoadString with Mac STR#
    ( IL_HINST hinst,             // Module "instance" handle for the Mac
      UINT16 nIdResource,          // Windows-style string resource ID
      IL_PSTR lpBuffer,            // Pointer to buffer to receive the string
      UINT16 nBuffer );           // Length of caller's buffer

//----- Macintosh functions used to implement ILMACRO file I/O macros
int IL_DECL MACFile_GetDir         // Retrieve the default directory path name
    ( IL_PSTR pPath );           // Pointer to string to receive path name
int IL_DECL MACFile_SetDir         // Set the default directory by path name
    ( IL_PSTR pPath );           // Pointer to the directory path name
int IL_DECL MACFile_MakeDir        // Create a new directory by path name
    ( IL_PSTR pPath );           // Pointer to the directory path name
int IL_DECL MACFile_Exists         // Determine if file/directory exists
    ( IL_PSTR pPath );           // Pointer to file/directory path name
int IL_DECL MACFile_SetEOF         // Set file logical length
    ( IL_PSTR pPath,             // Pointer to file path name
      long lLength );           // New file length in bytes
int IL_DECL MACFile_Length         // Get file/directory (logical) length
    ( IL_PSTR pPath,             // Pointer to file/directory path name
      long *plLength );         // Pointer to variable to receive length
int IL_DECL MACFile_Attrib         // Get file/directory (logical) length
    ( IL_PSTR pPath,             // Pointer to file/directory path name
      int *piAttribs );         // Pointer to variable to receive length
int IL_DECL MACFile_Info           // Get file/directory attributes and length
    ( IL_PSTR pPath,             // Pointer to file/directory path name
      int *piAttribs,            // Pointer to variable to receive attributes
      long *plLength );         // Pointer to variable to receive length
int IL_DECL MACFile_SetType        // Set Mac file type and creator
    ( IL_PSTR pPath,             // Pointer to file path name
      long lType,                // File type code (e.g., 'TEXT')
      long lCreator );          // File creator code (e.g., 'MPS ')
IL_PSTR IL_DECL MACFile_AppPath    // Get current application path name
    ( IL_PSTR pPath );           // Pointer to location to store path name
IL_PSTR IL_DECL MACFile_AppDir     // Get current application directory path
    ( IL_PSTR pDir );           // Pointer to location to store dir name
IL_PSTR IL_DECL MACFile_GetILDir   // Get pathname of IL translation directory
    ( IL_PSTR pDir );           // Pointer to location to store dir name
IL_PSTR MACFile_AppFile            // Get current application file name
    ( IL_PSTR pFile );           // Pointer to location to store file name
void IL_DECL MACFile_SplitPath     // Split a path name into components

```

```

    ( IL_PSTR pPath,
      IL_PSTR pDrive,
      IL_PSTR pDir,
      IL_PSTR pName,
      IL_PSTR pExt );
void IL_DECL MACFile_MakePath
    ( IL_PSTR pPath,
      IL_PSTR pDrive,
      IL_PSTR pDir,
      IL_PSTR pName,
      IL_PSTR pExt );
int MACFile_Open
    ( IL_PSTR path,
      int openMode,
      IL_HFILE *theFileNum );
int MACFile_Close
    ( IL_HFILE hFile );
int MACFile_Read
    ( IL_HFILE hFile,
      char *buffer,
      int count,
      int *readCount );
int MACFile_Write
    ( IL_HFILE hFile,
      char *buffer,
      int count );
int MACFile_Seek
    ( IL_HFILE hFile,
      unsigned long position,
      int mode);
long MACFile_Size
    ( IL_HFILE hFile );
long MACFile_Tell
    ( IL_HFILE hFile );
IL_PSTR IL_DECL MACFile_TempName
    ( IL_PSTR pDir,
      IL_PSTR pPrefix,
      IL_PSTR pPath,
      BOOLEAN bCreate );
int IL_DECL MACFile_Copy
    ( char *pszSourcePath,
      char *pszTargetPath );

//----- Functions used by MACFILE.C. May be of some value externally.
IL_PSTR IL_DECL GetPathFromFSSpec
    ( FSSpec* spec,
      IL_PSTR Path);
void IL_DECL pstrcpy
    ( Str255 dst,
      Str255 src);
void IL_DECL pstrcat
    ( Str255 dst,
      Str255 src);
void IL_DECL pstrinsert
    ( Str255 dst,
      Str255 src);

//----- Mac functions used to implement IL_ALLOC, IL_REALLOC, etc. macros
IL_HANDLE IL_DECL MacMem_Alloc
    ( UINT32 nBytes );
void IL_DECL MacMem_Free
    ( IL_HANDLE hBlock );
void IL_DIST *IL_DECL MacMem_Lock
    ( IL_HANDLE hBlock );
IL_HANDLE IL_DECL MacMem_Realloc
    ( UINT32 nBytes,
      IL_HANDLE hBlock );
void IL_DECL MacMem_Unlock
    ( IL_HANDLE hBlock );
void IL_DECL MacMem_Xlate
    ( BOOLEAN bXlate );

//----- Mac functions used to implement malloc, realloc, free, new, and delete
void * MacUT_malloc
    ( size_t nBytes );

```

```

void * MacUT_calloc                // Allocate/lock/clear relocatable memory
( size_t nItems,                  // Number of items to allocate
  size_t nBytesPerItem );        // Number of bytes per item
void * MacUT_realloc              // Grows or shrinks a block of memory
( void * pMemory,                // Pointer to the existing memory block
  size_t nBytes );               // New size for reallocated memory block
void MacUT_free                   // Frees a relocatable block of memory
( void * pMemory );              // Pointer to the existing memory block

//----- Macintosh "DLL" (ICRZ code resource) support functions
IL_HINST ILMAC_LoadLibrary        // Load code resource library by name
( IL_PSTR lpLibName );           // Name of code resource req'd
void ILMAC_FreeLibrary            // Unload code resource and free resources
( IL_HINST hinst );              // code resource info
long ILMAC_CallCodeResource       // Call a function in an ICRZ code resource
( IL_HINST hInst,                // hInst points to ICRZ "header" struct
  long nFunction,                // Function code of the requested function
  IL_PANY pParam );              // pointer to function parameter block

//----- Macintosh functions for implementing Windows "profiles" on the Mac
int IL_DECL
  IL_GetPrivateProfileString      // Get profile value from ILDF file
  ( IL_PSTR pszSection,          // Profile "Section" name
    IL_PSTR pszVariable,         // Profile "Variable" name
    IL_PSTR pszDefault,          // Default "Value" if variable not found
    IL_PSTR pszValue,            // Pointer to string to receive the value
    int nValLength,              // Length of caller's value string buffer
    IL_PSTR pszFileName );        // Profile "File" name
BOOLEAN IL_DECL
  IL_WritePrivateProfileString    // Write profile value into ILDF file
  ( IL_PSTR pszSection,          // Profile "Section" name
    IL_PSTR pszVariable,         // Profile "Variable" name
    IL_PSTR pszValue,            // Profile "Value" string
    IL_PSTR pszFileName );        // Profile "File" name
int IL_DECL IL_DeleteProfile      // Delete all records for specific FileName
( IL_PSTR pszFileName );          // Profile "File" name
int IL_DECL IL_DeleteProfileString // Delete profile string
( IL_PSTR pszFileName,          // Profile "File" name
  IL_PSTR pszSection,           // Profile "Section" name
  IL_PSTR pszVariable );        // Profile "Variable" name
int IL_DECL IL_GetProfileString   // Get profile value from ILDF file
( IL_PSTR pszSection,          // Profile "Section" name
  IL_PSTR pszVariable,         // Profile "Variable" name
  IL_PSTR pszDefault,          // Default "Value" if variable not found
  IL_PSTR pszValue,            // Pointer to string to receive the value
  int nValLength );             // Length of caller's value string buffer
int IL_DECL
  IL_GetPrivateProfileInt        // Get profile integer from ILDF file
  ( IL_PSTR pszSection,          // Profile "Section" name
    IL_PSTR pszVariable,         // Profile "Variable" name
    int nDefault,                // Default "Value" if variable not found
    IL_PSTR pszFileName );        // Profile "File" name
int IL_DECL IL_GetProfileInt      // Get profile integer from ILDF file
( IL_PSTR pszSection,          // Profile "Section" name
  IL_PSTR pszVariable,         // Profile "Variable" name
  int nDefault );                // Default "Value" if variable not found
BOOLEAN IL_DECL
  IL_WriteProfileString          // Write profile value into ILDF file
  ( IL_PSTR pszSection,          // Profile "Section" name
    IL_PSTR pszVariable,         // Profile "Variable" name
    IL_PSTR pszValue );          // Profile "Value" string

#endif // ILMAC

/*-----
 * The following definitions are for a general-purpose buffer management
 * mechanism, originally written by Bob Daley as ILXNOW\BUFFER.H.
 *-----*/

typedef struct                    // Buffer "header" structure
{
  IL_PANY    pBuffer;            // Pointer to buffer, NULL == not allocated
  IL_HANDLE  hBuffer;            // Handle for buffer, if pBuffer != NULL
  long       lBufSize;           // Buffer size currently allocated
  long       lMaxSize;           // Size beyond which buffer must not grow

```

```

    long          lGrowthIncrement;    // Number of bytes to (re-)allocate. Zero
                                        // .. to allocate just enough to fit
    IL_HANDLE     hBufferHeader;       // Added 3/24/96. See note below.
} ILUT_BUFFER, IL_DIST * ILUT_PBUFFER;

/*-----
 * NOTE: the 'hBufferHeader' member of ILUT_BUFFER is a johnny-come-lately
 * which must be used with caution to avoid backward compatibility issues.
 * ILTR_pTmpBuf and ILTR_pSSTBuf were introduced long before, so the handles
 * for those structures are stored in the "tr" structure, not here.
 *-----*/

int ILUT_InitBuffer          // Initialize a buffer
( ILUT_PBUFFER phHeader,
  long lInitialSize,        // initial size; zero for no initial allocation
  long lGrowthIncrement,    // minimum growth; zero for "just enough"
  long lMaxSize );          // max size beyond which buffer will not grow

int ILUT_GetBuffer           // (Re)-allocates buffer of needed size
( ILUT_PBUFFER phHeader,    // Pointer to buffer header structure
  long lBytesNeeded );      // Number of bytes needed

void ILUT_FreeBuffer         // Frees buffer, IF it has been allocated
( ILUT_PBUFFER phHeader );  // Pointer to buffer header structure

#ifdef __cplusplus           // Special handling for C++
}
#endif // __cplusplus

#endif // __ILUTIL

```

```

/*-----
* Module:    BUFFER.C
* Purpose:   Buffer management functions.
* Functions:
*           ILUT_InitBuffer - Initialize a buffer
*           ILUT_GetBuffer -- Get a pointer to a buffer of specified size.
*                           Allocates or reallocates the buffer as needed.
*           ILUT_FreeBuffer - Free buffer if allocated and zero both the
*                           buffer handle and buffer pointer.
* Author:    Bob Daley, Copyright (c) IntelliLink Corporation, 1995
*-----*/

#include "ilutil.h"

/*-----
* Function:   ILUT_InitBuffer
* Purpose:    Initialize a buffer
* Returns:    SUCCESS or error code
*-----*/
int ILUT_InitBuffer
(
    ILUT_PBUFFER phHeader,
    long lInitialSize,    // initial size; zero for no initial allocation
    long lGrowthIncrement, // minimum growth; zero for "just enough"
    long lMaxSize )      // max size beyond which buffer will not grow
{
    int rc;

    if (lGrowthIncrement > lMaxSize)
        return ILUT_ERR_SIZE;

    //----- zero out the header
    phHeader->pBuffer = NULL;
    phHeader->hBuffer = IL_NULL_HANDLE;
    phHeader->lBufSize = 0;
    phHeader->lMaxSize = lMaxSize;

    if (lInitialSize == 0)
        rc = SUCCESS;
    else
    {
        //----- Allocate the buffer to its initial size - EXACTLY!
        phHeader->lGrowthIncrement = 0;
        rc = ILUT_GetBuffer (phHeader, lInitialSize);
    }

    phHeader->lGrowthIncrement = lGrowthIncrement;

    return rc;
}

/*-----
* Function:   ILUT_GetBuffer
* Purpose:    Get a pointer to a buffer of specified size. Allocates or
*             reallocates the buffer as needed.
* Input:      phHeader ----- Pointer to buffer header structure
*             lBytesNeeded --Number of bytes actually needed
*
* NOTE:       buffer is never allowed to grow beyond max size. If user asks
*             for more than the max he'll just get the max.
*
* Returns     SUCCESS or one of the following error codes:
*             ILUT_ERR_NOMEM -- allocation failure
*             ILUT_ERR_SETTLE_FOR_LESS -- you asked for more than max; we're
*             just giving you the max
*-----*/
int ILUT_GetBuffer
(
    ILUT_PBUFFER phHeader,    // (Re)-allocates buffer of needed size
    long lBytesNeeded )       // Pointer to buffer header structure
{
    long lBytes;              // Number of bytes to be allocated
    int BestPossibleResult;

```



```

//----- If buffer not allocated, allocate the buffer now.
if (phHeader->pBuffer == NULL)
{
    //----- Try to allocate the buffer for the first time.
    lBytes = max (lBytesNeeded, phHeader->lGrowthIncrement);
    IL_ALLOC_MEM (lBytes, phHeader->hBuffer, phHeader->pBuffer);
    if (phHeader->pBuffer == NULL)
        return ILUT_ERR_NOMEM;

    //----- Allocation successful. Set current length and return pointer.
    phHeader->lBufSize = lBytes;
    return SUCCESS;
}

//----- Buffer already allocated. If it's big enough, we're all done.
if (phHeader->lBufSize >= lBytesNeeded)
    return SUCCESS;

//----- Check to see if user is asking for more than the max
if (lBytesNeeded > phHeader->lMaxSize)
    BestPossibleResult = ILUT_ERR_SETTLE_FOR_LESS;
else
    BestPossibleResult = SUCCESS;

//----- Buffer too small. Determine how much to grow buffer
lBytes = max (lBytesNeeded, phHeader->lBufSize + phHeader->lGrowthIncrement);

//----- Never grow buffer beyond MAX size
lBytes = min (lBytes, phHeader->lMaxSize);

//----- Try to reallocate buffer to the larger size
IL_REALLOC_MEM (lBytes, phHeader->hBuffer, phHeader->pBuffer);
if (phHeader->pBuffer == NULL)
    return ILUT_ERR_NOMEM;

//----- Re-allocation successful. Adjust length.
phHeader->lBufSize = lBytes;

//----- Return SUCCESS or ILUT_ERR_SETTLE_FOR_LESS
return BestPossibleResult;
}

/*-----
* Function:    ILUT_FreeBuffer
* Purpose:    Free buffer if allocated and zero buffer handle and pointer
* Input:      phHeader ----- Pointer to buffer header structure
* Returns     void
*-----*/

void ILUT_FreeBuffer          // Frees buffer, IF it has been allocated
( ILUT_PBUFFER phHeader )    // Pointer to buffer header structure
{
    //----- Free the buffer if it has been allocated
    if (phHeader->pBuffer != NULL)
        IL_FREE_MEM (phHeader->hBuffer, phHeader->pBuffer);

    //----- Always clear the pointer, handle, and currently allocated length
    phHeader->pBuffer = NULL;
    phHeader->hBuffer = IL_NULL_HANDLE;
    phHeader->lBufSize = 0;
}

```



```

#if !defined(__ILTBL)

/*-----*/
* Name:      ILTBL.H
* Purpose: Header file for table handling
* Author:   Mike Blanchette, Copyright (c) IntelliLink Corporation, 1993
*-----*/
#define __ILTBL

//----- Dependent include files
#ifdef ILWIN
#include <windows.h>
#endif
#include <stdlib.h>
#include "ilx.h"
#include "ilutil.h"

//----- Limits
#define ILTB_MAX_SYS_FREE      40          // System free space
#define ILTB_MAX_SEC_FREE     18          // Section free space
#define ILTB_MAX_CHECKS       20          // Max number of checks
#define ILTB_MAX_COMPORTS      9          // Max number of com ports
#define ILTB_MAX_TYPENAME     20          // Max size of type names
#define ILTB_MAX_LINE         80          // Max length of text line
#define ILTB_MAX_MSG          200         // Max length of message
#define ILTB_MAX_NAME         31          // Size of name field
#define ILTB_NUM_FLDTYPES      7          // Number of field types
#define ILTB_NUM_SECTIONS     10          // Number of section types
#define ILTB_NUM_SPECIAL      64          // Number of DOS special chars

//----- Platform dependent limits
#ifdef ILWIN
#ifdef _WIN32
// WIN32 definitions
#define ILTB_MAX_PATH          MAX_PATH   // Size of path name
#define ILTB_MAX_DIR           MAX_DIR    // Size of directory name
#define ILTB_MAX_FILES         5          // Number of file names saved
#define ILTB_MAX_DRVNAME       9          // Size of xlator driver name
#define ILTB_MAX_EXT           30         // Size of file extension list
#define ILTB_MAX_SLOT          65         // Size of file slot in SYSREC
#else
// WIN16 definitions
#define ILTB_MAX_PATH          65         // Size of path name
#define ILTB_MAX_DIR           65         // Size of directory name
#define ILTB_MAX_FILES         5          // Number of file names saved
#define ILTB_MAX_DRVNAME       9          // Size of xlator driver name
#define ILTB_MAX_EXT           30         // Size of file extension list
#define ILTB_MAX_SLOT          65         // Size of file slot in SYSREC
#endif
#endif
#else
#define ILTB_MAX_PATH          MAX_PATH   // Size of path name
#define ILTB_MAX_DIR           MAX_DIR    // Size of directory name
#define ILTB_MAX_FILES         1          // Number of file names saved
#define ILTB_MAX_DRVNAME       9          // Size of xlator driver name
#define ILTB_MAX_EXT           30         // Size of file extension list
#define ILTB_MAX_SLOT          MAX_PATH   // Size of file slot in SYSREC
#endif
#endif // ILWIN

//----- INI file constants
#define ILTB_INI_CDFMAP        "CDFMAP"    // Use only mapped CDF fields
#define ILTB_INI_CDFNAMES      "CDFNAMES"  // First CDF record names
#define ILTB_INI_CDFSEP        "CDFSEP"    // CDF separator
#define ILTB_INI_CHECKED       "CHECKED"   // Checked section
#define ILTB_INI_COMMENT       "; IntelliLink options"
#define ILTB_INI_COMPORT       "COMPORT"   // Communication port
#define ILTB_INI_CONFIRM       "CONFIRM"   // Show confirmation dialog
#define ILTB_INI_DEFFIELDS     "ILDEFFIELDS" // Field resource name
#define ILTB_INI_DEFLOG        "IL.LOG"    // Default log file name
#define ILTB_INI_DISCON        "DISCONNECT" // Disconnect after merge
#define ILTB_INI_EDTFILE       "EDITOR"    // Text editor name
#define ILTB_INI_DEVTYPE       "DEVICE"    // Device type
#define ILTB_INI_KEEPLLOG      "KEEPLLOG"  // Keep log file
#define ILTB_INI_EXIT          "EXITAFTER" // Exit after translation
#define ILTB_INI_FILE          "ILWIN.INI" // Name of INI file
#define ILTB_INI_HIDE          "HIDE"      // Hide main window
#define ILTB_INI_LOGFILE       "LOGFILE"   // Log file name
#define ILTB_INI_NEW           "CONFIRMNEW" // Confirm new file

```

```

#define ILTB_INI_NOTEPAD      "NOTEPAD.EXE" // Default editor name
#define ILTB_INI_NOTSET      "NO" // Option is OFF
#define ILTB_INI_OVERWRITE   "CONFIRMOVER" // Confirm overwrite
#define ILTB_INI_PRISYS      "PRISYS" // Primary system
#define ILTB_INI_PURGELOG    "PURGELOG" // Always purge log
#define ILTB_INI_SAVEWORK    "SAVEWORK" // Save work files
#define ILTB_INI_SECSYS      "SECNSYS" // Secondary system
#define ILTB_INI_SECTION     "ILWIN" // INI section file
#define ILTB_INI_SET         "YES" // Option is ON
#define ILTB_INI_WARNMERGE   "WARNMERGE" // Warn user before merge

//----- Table names
#define ILTB_CON_TBL         "CONFIG.IL" // Configuration table
#define ILTB_FLD_TBL         "FIELDS.IL" // Field map table
#define ILTB_FLT_TBL         "FILTERS.IL" // Filters table
#define ILTB_SYS_TBL         "SYSTEM.IL" // System table
#define ILTB_TABLES_FILE    "TABLES.ITB" // Combined ILTB tables file

//----- Initial and increment table sizes
#define ILTB_DEF_CON_TBL     4000 // Default Configuration index
#define ILTB_INC_CON_TBL     250 // Increment size
#define ILTB_DEF_FLD_TBL     4000 // Default Fields index
#define ILTB_INC_FLD_TBL     250 // Increment size
#define ILTB_DEF_FLT_TBL     100 // Default Filters index
#define ILTB_INC_FLT_TBL     50 // Increment size
#define ILTB_DEF_SYS_TBL     100 // Default System index
#define ILTB_INC_SYS_TBL     50 // Increment size

//----- Error constants
#define ILTB_ERR_NOINI       1 // Unable to find INI file
#define ILTB_ERR_IO          2 // IO error encountered
#define ILTB_ERR_BADOPT      3 // Invalid INI file option

//----- General data types
typedef ILX_ACCESS ILTB_ACC; // Access type
typedef ILX_RANGE ILTB_APPT; // Appointment Range type
typedef UINT32 ILTB_ATTRIB; // Attribute Type
typedef INT16 ILTB_CLASS; // System class type
typedef INT16 ILTB_FILE; // File attribute type
typedef INT16 ILTB_ID; // ID Type
typedef ILTB_ATTRIB * ILTB_PATTRIB; // Pointer to Attribute Type
typedef ILX_OPTION ILTB_REC; // Reconciliation Option type
typedef INT16 ILTB_SEC; // Section type
typedef INT32 ILTB_TAG; // Configuration Tag
typedef ILX_RANGE ILTB_TODO; // Todo Range type
typedef INT16 ILTB_TYPE; // System Type

//----- Access types
#define ILTB_ACC_NONE        ILX_ACCESS_NONE // Not applicable
#define ILTB_ACC_DDE         ILX_ACCESS_DDE // DDE
#define ILTB_ACC_DBASE       ILX_ACCESS_DBASE // dBASE
#define ILTB_ACC_FILE        ILX_ACCESS_FILE // File
#define ILTB_ACC_PDX         ILX_ACCESS_PDX // Paradox
#define ILTB_ACC_CDF         ILX_ACCESS_CDF // CDF
#define ILTB_ACC_ODBC        ILX_ACCESS_ODBC // ODBC database
#define ILTB_ACC_NEW1        ILX_ACCESS_NEW1 // Unassigned
#define ILTB_ACC_NEW2        ILX_ACCESS_NEW2 // Unassigned

//----- Appointment Range
#define ILTB_APPT_ALL         ILX_RANGE_ALL // All appointments
#define ILTB_APPT_FUTURE      ILX_RANGE_FUTURE // Future-dated appointments
#define ILTB_APPT_NONE        ILX_RANGE_NONE // Not applicable
#define ILTB_APPT_DEFAULT     ILX_RANGE_DEFAULT // Use system default

//----- General attributes
#define ILTB_ATT_ILWIN        0x00000001L // Running under ILWIN
#define ILTB_ATT_ACTIVE       0x00000002L // Active
#define ILTB_ATT_WINPAD       0x00000004L // Running under WinPad
#define ILTB_ATT_LITE         0x00000008L // Running under Lite

//----- Application attributes
#define ILTB_ATT_VAR_SECTS    0x00000010L // Variable number of sections
#define ILTB_ATT_INSERT       0x00000020L // Insert-capable application
#define ILTB_ATT_UPDATE       0x00000040L // Update-capable application
#define ILTB_ATT_NOTIFY       0x00000080L // Notify-capable application

```

```

#define ILTB_ATT_MERGE          0x00000100L    // Merge-capable application
#define ILTB_ATT_TODO          ILX_ATT_TODO     // Todo-capable application
#define ILTB_ATT_GROUPS        ILX_ATT_GROUPS   // Groups-capable application
#define ILTB_ATT_SELKEY        0x00000800L    // User-selectable key field
#define ILTB_ATT_FORCENOTE     0x00001000L    // Always force Note field
#define ILTB_ATT_DOCNAME       0x00002000L    // Document rather than file
#define ILTB_ATT_NOSELECT      0x00004000L    // Disable file selection
#define ILTB_ATT_NOIMPORT      0x00008000L    // Translator can only export
#define ILTB_ATT_ZOOMEXT       ILX_ATT_ZOOMEXT  // Zoomer extensions
#define ILTB_ATT_DISCONN       0x00020000L    // Always force disconnection
#define ILTB_ATT_SINGLEFILE    0x00040000L    // All sections in one file
#define ILTB_ATT_REMFILE       0x00080000L    // Remove target file before import
#define ILTB_ATT_RAMCARD       0x00100000L    // Use RAM card (engine only)
#define ILTB_ATT_SHOWNONE      0x00200000L    // Show NONE reconcile option
#define ILTB_ATT_INSTREM       0x00400000L    // Remove install required?
#define ILTB_ATT_ILXWIN_32     0x00800000L    // Runs only under ILX/Win32
#define ILTB_ATT_HASOPTIONS    0x01000000L    // System has options
#define ILTB_ATT_SYNC_CAPABLE  0x02000000L    // System can handle "SyncPort"
#define ILTB_ATT_TARMUSTEXIST  0x04000000L    // Target file must exist
#define ILTB_ATT_MUST_FAN_B4U  0x08000000L    // Must Fan Before Unload
#define ILTB_ATT_TOTAL_REBUILD 0x10000000L    // Must Re-Build App DB to change it
#define ILTB_ATT_SYSNEW_6      0x20000000L    // Attribute not yet assigned
#define ILTB_ATT_SYSNEW_7      0x40000000L    // Attribute not yet assigned
#define ILTB_ATT_SYSNEW_8      0x80000000L    // Attribute not yet assigned

```

//----- Configuration attributes

```

#define ILTB_ATT_SELECTED      0x00000010L    // Selected configuration
#define ILTB_ATT_REVERSED      0x00000020L    // Systems are reversed
#define ILTB_ATT_CHANGED       0x00000040L    // Configuration has changed
#define ILTB_ATT_BASECON       0x00000080L    // Template configuration
#define ILTB_ATT_NORMCON       0x00000100L    // Normal configuration

```

//----- Field attributes

```

#define ILTB_ATT_FLOAT          0x00000010L    // Floating field attribute
#define ILTB_ATT_VIEW           0x00000020L    // Display field attribute
#define ILTB_ATT_DEFAULT       ILX_ATT_DEFAULT // Default phone number
#define ILTB_ATT_COMBINED      0x00000080L    // Combined field
#define ILTB_ATT_NAME          0x00000100L    // Default Name field
#define ILTB_ATT_COMPANY       0x00000200L    // Default Company field
#define ILTB_ATT_DATEFLD       0x00000400L    // Appointment Date field
#define ILTB_ATT_DONEFLAG      ILTB_ATT_DATEFLD // To Do done flag: see note below
#define ILTB_ATT_MUSTMAP       0x00000800L    // Require field to be mapped
#define ILTB_ATT_WARNMAP       0x00001000L    // Warn if field is not mapped
#define ILTB_ATT_MULTLINE      0x00002000L    // Multi-line field
#define ILTB_ATT_KEEPHONE      0x00004000L    // Keep all phone components
#define ILTB_ATT_ALARMRELATED  0x00008000L    // Alarm-related field (eg a Date)
#define ILTB_ATT_STARTDATETIME 0x00010000L    // START date or time (see type)
#define ILTB_ATT_ENDDATETIME   0x00020000L    // END date or time (see type)
#define ILTB_ATT_KEY_FIELD     0x00040000L    // Key field for reconciliation
#define ILTB_ATT_HASH_FIELD    0x00080000L    // Hash code this field for lookup
#define ILTB_ATT_VAL_REQUIRED  0x00100000L    // Field must have a value
#define ILTB_ATT_NO_RECONCILE  0x00200000L    // No reconciliation on this field
#define ILTB_ATT_PRIORITY      0x00400000L    // Field is a "priority" field
#define ILTB_ATT_FUNCTIONCODE  0x00800000L    // e.g. Sharp TEL1
#define ILTB_ATT_HIDDEN_FIELD  0x01000000L    // Don't display field to user
#define ILTB_ATT_TAGGED        0x02000000L    // Put IL "decorations" here
#define ILTB_ATT_INSENSITIVE   0x04000000L    // for case-independent compares
#define ILTB_ATT_COMPARE_STRIPPED 0x08000000L // strip before comparing
#define ILTB_ATT_FLDNEW_3      0x10000000L    // Attribute not yet assigned
#define ILTB_ATT_FLDNEW_4      0x20000000L    // Attribute not yet assigned
#define ILTB_ATT_FLDNEW_5      0x40000000L    // Attribute not yet assigned
#define ILTB_ATT_FLDNEW_6      0x80000000L    // Attribute not yet assigned

```

/*-----

* Note: the field list for every "MAIN" section (i.e. a section that has
 * subtype==ILX_SUBSECT_MAIN==0) should have exactly one field that has
 * the ILTB_ATT_TAGGED attribute. In addition to holding "real" native
 * data, a "TAGGED" field may contain an IntelliLink-made TAG.

* In particular the section subtype code is stored in this field.

* For example if description="Staff Meeting" and subtype=23, we may
 * construct a "decorated" description of "Staff Meeting {23}".

* The ILTB_ATT_TAGGED attribute is attached to an ordinary

```

*      user-visible field, not to the hidden field called "_subType".
*
* Note: ILTB_ATT_DONEFLAG is intentionally defined to ILTB_ATT_DATEFLD.
* The reuse of the bit is possible because these fields will never exist
* in the same section.
*-----*/

//----- Map attributes
#define ILTB_ATT_BASEMAP      0x000000010L    // Template field map
#define ILTB_ATT_NORMAP      0x000000020L    // Normal field map

//----- Section attributes
#define ILTB_ATT_PAD_NAMES    0x000000010L    // Pad file names
#define ILTB_ATT_TEMPLATE    0x000000020L    // Section template
#define ILTB_ATT_EDIT_LABELS  0x000000040L    // User can change field names
#define ILTB_ATT_ADD_FIELDS   0x000000080L    // User can add new fields
#define ILTB_ATT_ADD_SECTS    0x000000100L    // User can add section names
#define ILTB_ATT_PCCARD_RAM   0x000000200L    // Use PC Card RAM vs Main
#define ILTB_ATT_WARNMERGE    0x000000400L    // Warn user on merge
#define ILTB_ATT_ASKFIELDS    0x000000800L    // Ask translator for field list?
#define ILTB_ATT_TWOPASS      0x000001000L    // Two-pass translation
#define ILTB_ATT_NOFANNING    0x000002000L    // Do not fan repeating items
#define ILTB_ATT_NOMAPPING    0x000004000L    // Does not support field mapping
#define ILTB_ATT_SEC_PARENT   0x000008000L    // Section is a "parent" section
#define ILTB_ATT_SEC_CHILD    0x000010000L    // Section is a "child" section
#define ILTB_ATT_SEC_USESRC    0x000020000L    // Use source fields in new file
#define ILTB_ATT_DEL_OUTRANGE 0x000040000L    // SYNC: delete out-of-range items
#define ILTB_ATT_RENAME_SECTS 0x000080000L    // User can rename sections
#define ILTB_ATT_SECNEW_2     0x000100000L    // Attribute not yet assigned
#define ILTB_ATT_SECNEW_3     0x000200000L    // Attribute not yet assigned
#define ILTB_ATT_SECNEW_4     0x000400000L    // Attribute not yet assigned

//----- Reconciliation options
#define ILTB_REC_REPLACE      ILX_OPT_REPLACE  // Replace conflicting items
#define ILTB_REC_IGNORE      ILX_OPT_IGNORE   // Ignore conflicting items
#define ILTB_REC_NOTIFY      ILX_OPT_NOTIFY   // Notify user of conflicts
#define ILTB_REC_INSERT      ILX_OPT_INSERT   // Add conflicting items
#define ILTB_REC_UPDATE      ILX_OPT_UPDATE   // Update existing items
#define ILTB_REC_MERGE       ILX_OPT_MERGE    // Merge new items (Sharp)
#define ILTB_REC_CANCEL      ILX_OPT_ACCEPT_1 // Cancel translation
#define ILTB_REC_VERIFY      ILX_OPT_ACCEPT_2 // Recheck for conflicts
#define ILTB_REC_NONE        ILX_OPT_NONE     // No reconciliation
#define ILTB_REC_DELETE      ILX_OPT_DELETE   // Remove item
#define ILTB_REC_DELTA_ACK    ILX_OPT_DELTA_ACK // a FastSync Unload Action Code

//----- Section Types
#define ILTB_SEC_APPT         0                // Appointment
#define ILTB_SEC_DB           1                // Database
#define ILTB_SEC_DOCUMENT     2                // Word processing document
#define ILTB_SEC_MEMO         ILTB_SEC_DOCUMENT // Memo (now WP document)
#define ILTB_SEC_PHONE        3                // Phone Book
#define ILTB_SEC_TODO         4                // To-Do List
#define ILTB_SEC_GROUPS       5                // Phone Groups
#define ILTB_SEC_OUTLINE      6                // Outline
#define ILTB_SEC_CALL         7                // Calls
#define ILTB_SEC_SPREAD       8                // Spreadsheet
#define ILTB_SEC_EXPENSE      9                // Expense

//----- Todo Range
#define ILTB_TODO_ALL         ILX_RANGE_ALL     // All items
#define ILTB_TODO_NOTDONE     ILX_RANGE_FUTURE // Incomplete items
#define ILTB_TODO_NONE        ILX_RANGE_NONE    // Not applicable
#define ILTB_TODO_DEFAULT     ILX_RANGE_DEFAULT // Use system default

//----- System Class
#define ILTB_CLASS_ANY        1                // Any mapping supported
#define ILTB_CLASS_EXCLUDE    2                // Exclude mapping to self
#define ILTB_CLASS_HPPIM      3                // "Cougar" application
#define ILTB_CLASS_SIMON      4                // Simon phone
#define ILTB_CLASS_BOSS       5                // Casio BOSS family
#define ILTB_CLASS_WIZARD     6                // Sharp Wizard family
#define ILTB_CLASS_PSION      7                // Psion family

//----- File Attributes
#define ILTB_FILE_LEFT        0x0001           // Left file is present

```

```

#define ILTB_FILE_RIGHT      0x0002      // Right file is present

//----- System Types
#define ILTB_TYPE_APP        0           // Desktop application
#define ILTB_TYPE_CASIO_1    1           // Casio Series 4000-5000
#define ILTB_TYPE_CASIO_2    2           // Casio Series 7000-9000
#define ILTB_TYPE_HP95LX     3           // HP 95LX
#define ILTB_TYPE_HP100LX    4           // HP 100LX
#define ILTB_TYPE_PSION      5           // Psion Series 3
#define ILTB_TYPE_SHARP_1    6           // Sharp Series 5000-7000
#define ILTB_TYPE_SHARP_2    7           // Sharp Series 7600/7620
#define ILTB_TYPE_SHARP_3    8           // Sharp Series 8000-8200
#define ILTB_TYPE_SHARP_4    9           // Sharp Series 8600/YO-610
#define ILTB_TYPE_SHARP_5   10           // Sharp Series 9600
#define ILTB_TYPE_ZOOMER_1   11           // Zoomer "DOS"
#define ILTB_TYPE_ZOOMER_2   12           // Zoomer "native"
#define ILTB_TYPE_FRANKLIN   13           // Franklin's Tower
#define ILTB_TYPE_ZAURUS     14           // Sharp Zaurus
#define ILTB_TYPE_CASIO_3    15           // Casio SF-R10/20
#define ILTB_TYPE_CASIO_4    16           // Casio 5300B
#define ILTB_TYPE_CASIO_5    17           // Casio 7900
#define ILTB_TYPE_CASIO_6    18           // Casio CSF-7950 (color)
#define ILTB_TYPE_CASIO_7    19           // Casio NX-4000 (pen)
#define ILTB_TYPE_SHARP_6    20           // Sharp 5500
#define ILTB_TYPE_OMNIGO_100 21           // HP Omnigo 100
#define ILTB_TYPE_NEW_3      22           // Future expansion
#define ILTB_TYPE_NEW_4      23           // Future expansion

/*-----
 * >>> Note that when you extend the above list, be sure to update
 * >>> the integrity check in the tablecl utility!!!!
 *-----*/

//----- Field Map Table data types
typedef struct {                          // Field index entry
    ILTB_ID appID;                        // Application ID
    ILTB_ID sectionID;                    // Section ID
    INT16 fieldNum;                       // Field count
    INT16 listSum;                        // List CHECKSUM value
    INT32 listFset;                       // List offset in file
    char common;                          // Default field list?
} ILTB_NDX;

typedef struct {                          // Field index record
    INT16 listNum;                       // Number of field lists
    ILTB_NDX ndx[1];                     // Field index
} ILTB_FLDNDX;

typedef ILTB_FLDNDX IL_DIST *ILTB_PFLDNDX; // Pointer to field index

typedef struct {                          // Field list cross-reference
    ILTB_ID sourceID;                    // Source application ID
    ILTB_ID srcSectID;                   // Source section ID
    ILTB_ID targetID;                    // Target application ID
    ILTB_ID tarSectID;                   // Target section ID
    ILTB_ID mapID;                       // Map ID
} ILTB_XRF;

typedef struct {                          // Field xref record
    INT16 mapNum;                        // Total number of field maps
    ILTB_XRF xref[1];                   // Field map cross-reference
} ILTB_MAPXRF;

typedef ILTB_MAPXRF IL_DIST *ILTB_PMAPXRF; // Pointer to xref record

typedef struct {                          // Field map record
    ILTB_ID sourceList;                  // Source list index
    ILTB_ID targetList;                  // Target list index
    char name[ILX_MAX_FLDNAME];          // Name of field map
    UINT32 attribs;                      // Map attributes
    INT16 mapIndex[1];                   // Map index number
} ILTB_MAP;

typedef ILTB_MAP IL_DIST *ILTB_PMAP;      // Pointer to map record

```



```

typedef struct {
    INT16 fieldNum;
    ILTB_ID appID;
    ILTB_ID sectionID;
    char name[ILX_MAX_FLDNAME];
    UINT32 attribs;
    ILX_FIELD field[1];
} ILTB_FLDLST;

// Field list header
// Number of fields in list
// Application ID
// Section Type
// Field list name
// Field list attributes
// Field descriptor

typedef ILTB_FLDLST IL_DIST *ILTB_PFLDLST; // Pointer to field list record

//----- Section Table data types
typedef struct {
    ILTB_ID SectionID;
    ILTB_SEC SectionType;
    ILTB_ACC AccessType;
    char DefaultExt[ILTB_MAX_EXT];
    char PadChar;
    char Name[ILTB_MAX_NAME];
    char Code[ILTB_MAX_NAME];
    char File[ILTB_MAX_FILES][ILTB_MAX_SLOT];
    ILTB_ATTRIB FileAttrib;
    ILTB_ATTRIB SectionAttrib;
    INT16 SectionSubType;
    char filler[ILTB_MAX_SEC_FREE];
} ILTB_SECREC;

// Section Record
// Section ID
// Section Type
// Access Type
// Default File Extension
// File Name Pad Character
// Section name
// Section code
// File names
// File attributes
// Section Attributes
// Section Sub-type code
// Free space

typedef ILTB_SECREC IL_DIST *ILTB_PSECREC; // Pointer to Section Record

//----- System Table data types
typedef struct {
    INT16 SysClass;
    ILTB_ID SysID;
    ILTB_TYPE SysType;
    char SysName[ILTB_MAX_NAME];
    ILTB_ACC AccessType;
    char SecTitle[ILTB_MAX_NAME];
    char XlateName[ILTB_MAX_DRVNAME];
    char XporName[ILTB_MAX_DRVNAME];
    ILTB_REC ReconcileOpt;
    ILTB_APPT ApptRange;
    ILTB_TODO TodoRange;
    char AppLoc[ILTB_MAX_PATH];
    ILX32_16PAD(pad_01,195)
    INT16 ComPort;
    INT16 SectionType[ILTB_NUM_SECTIONS];
    ILTB_ATTRIB SysAttrib;
    ILTB_ID ImportCharMapID;
    ILTB_ID ExportCharMapID;
} ILTB_SYSREC;

// System Record
// System class
// System ID Number
// System Type
// System Name
// Access Type
// Section Title
// Translator Name
// Transporter Name
// Reconciliation Option
// Appointment Range
// To-Do Range
// Application Location
// Padding for long file names
// Communication Port
// Supported Section Types
// System Attributes
// Char map table ID for import
// Char map table ID for export

//----- Free space, must occur before Section below.
char filler[ILTB_MAX_SYS_FREE];

    INT16 SectionCount;
    ILTB_SECREC Section[1];
} ILTB_SYSREC;

// Section Count
// Sections

typedef ILTB_SYSREC IL_DIST *ILTB_PSYSREC; // Pointer to System Record

//----- Configuration Table data types
typedef struct {
    ILTB_ID ConfigID;
    ILTB_SEC SectionType;
    ILTB_TAG SourceTag;
    ILTB_TAG TargetTag;
    ILTB_ID MapID;
    ILTB_ID FilterID;
    ILTB_ATTRIB State;
} ILTB_CONREC;

// Configuration Record
// Configuration ID
// Section Type
// Source tag
// Target tag
// Field Map ID
// Filter ID
// Configuration State

typedef ILTB_CONREC IL_DIST *ILTB_PCONREC; // Pointer to Config Record

//----- INI file options
typedef struct {
    INT16 ComPort;
} ILTB_INI;

// INI file options
// Filer com port

```

```

char CDFSep;                // CDF separator character
char EditorName[_MAX_PATH]; // Name of text editor
char LogFileName[_MAX_PATH]; // Name of log file
BOOL16 CDFMapOnly;          // Use only mapped CDF fields?
BOOL16 CDFNames;            // First CDF record names?
BOOL16 Checked[ILTB_MAX_CHECKS]; // List of checked sections
BOOL16 ConfirmNew;          // Confirm file creation?
BOOL16 ConfirmOverwrite;    // Confirm file overwrite?
BOOL16 ConfirmXlate;        // Confirm before translation?
BOOL16 Disconnect;          // Disconnect after merge?
BOOL16 ExitAfterXlate;      // Exit after translation?
BOOL16 HideMain;            // Hide main window?
BOOL16 LogFile;             // Produce log file?
BOOL16 PurgeLog;            // Always purge log?
BOOL16 SaveWork;            // Save work files?
BOOL16 WarnMerge;           // Warn user before merge?
ILTB_ID PrimeSys;           // Primary system ID
ILTB_ID SecndSys;           // Secondary system ID
ILTB_TYPE DevType;          // Filer device type
} ILTB_OPTIONS;

typedef ILTB_OPTIONS IL_DIST *ILTB_POPTIONS; // Pointer to INI options

//----- Access macros
#define NDX_PRI_TAG(ndx, i) \
    ((ndx[i].State & ILTB_ATT_REVERSED) ? \
     ndx[i].TargetTag : ndx[i].SourceTag)

#define NDX_SND_TAG(ndx, i) \
    ((ndx[i].State & ILTB_ATT_REVERSED) ? \
     ndx[i].SourceTag : ndx[i].TargetTag)

//----- Function prototypes
int GetIniOptions          // Retrieve system options
    ( ILTB_POPTIONS option, // Pointer to options data
      IL_PSTR ilDir );      // Pointer to install directory
int PutIniOptions          // Write out system options
    ( ILTB_POPTIONS option, // Pointer to options data
      IL_PSTR ilDir );      // Pointer to install directory

#endif // __ILTBL

```

```

/*-----
* Name:      ILX.H
* Purpose: Header file for translator API
* Author: Copyright (c) IntelliLink Corporation, 1992-1995
*-----*/

#if !defined(__ILX)
#define __ILX // Signal header inclusion

//----- Make sure the resource compiler only sees the ILX_ERR codes
#ifndef RC_INVOKED

#include "iltypes.h" // Common IntelliLink types
#include "ilxtr.h"    // Definitions shared with ILTR

/*-----
* ILX header version control -- Please increment the version number by one
* for EVERY checked-in change to ILX.H -- Added 6/19/96
*-----*/

//----- Version number of this header to be updated for each checkin
#define ILX_CURRENT_VERSION 102 // corresponds to TLIB version

//----- Macros for testing ILX version number
#define ILX_VERSION_IS_AT_LEAST(ver) (ILXGL_version >= ver)
#define ILX_VERSION_IS_PRIOR_TO(ver) (ILXGL_version < ver)

//----- The resource compiler needs to see the following error codes
#endif // RC_INVOKED

/*-----
* Error codes.
*-----*/
#define ILX_OK 0 // Operation successful
#define ILX_NOTOK 1 // Operation failed
#define ILX_ERR_APP 2 // Invalid application handle
#define ILX_ERR_OPT 3 // Invalid reconciliation option
#define ILX_ERR_PIM 4 // Invalid application type
#define ILX_ERR_DOWN 5 // Translation engine shut down
#define ILX_ERR_UP 6 // Translation engine running
#define ILX_ERR_NOFILE 7 // Source file does not exist
#define ILX_ERR_NODIR 8 // Target dir does not exist
#define ILX_ERR_SAME 9 // Source is also target file
#define ILX_ERR_DUPSECT 10 // Duplicate section name
#define ILX_ERR_NOMEM 11 // Unable to allocate memory
#define ILX_ERR_FILES 12 // File names not specified
#define ILX_ERR_MAP 13 // Invalid field map ID
#define ILX_ERR_IO 15 // File IO error
#define ILX_ERR_FIELD 16 // Invalid field specification
#define ILX_ERR_RANGE 17 // Invalid appointment range
#define ILX_ERR_LOG 18 // Unable to access log file
#define ILX_ERR_ENGDLL 19 // Error accessing engine DLL
#define ILX_ERR_TRDLL 20 // Error accessing translate DLL
#define ILX_ERR_BOOL 21 // Invalid boolean value
#define ILX_ERR_VALTYPE 22 // Invalid value type
#define ILX_ERR_VALUE 23 // Invalid option value
#define ILX_ERR_MAPDLG 24 // Unable to show mapping dialog
#define ILX_ERR_CANCEL 25 // User has pressed CANCEL
#define ILX_ERR_SESSION 26 // Unable to get session handle
#define ILX_ERR_CODE 27 // Error code not found
#define ILX_ERR_MAPFILE 28 // Unable to access map file
#define ILX_ERR_NOAPPS 29 // No applications found
#define ILX_ERR_INVFILE 30 // Invalid file name
#define ILX_ERR_DBDLL 31 // Unable to load database DLL
#define ILX_ERR_PMODE 32 // Not running in protected mode
#define ILX_ERR_NODDE 33 // Unable to start DDE app
#define ILX_ERR_NODDEINIT 34 // Unable to initialize DDE
#define ILX_ERR_XLATE 35 // General translation error
#define ILX_ERR_OVERFLOW 36 // File overflow condition
#define ILX_ERR_BADFILE 37 // Unable to open or access file
#define ILX_ERR_ACTIVE 38 // Engine DLL already active
#define ILX_ERR_INACTIVE 39 // Engine DLL not active
#define ILX_ERR_COMDLL 40 // Unable to access comm DLL
#define ILX_ERR_CONNECT 41 // Unable to connect to device
#define ILX_ERR_COMPORT 42 // Unable to set com port

```



```

#define ILX_ERR_METERDLG          43    // Unable to show meter dialog
#define ILX_ERR_FILEGET           44    // Unable to get remote file
#define ILX_ERR_FILEPUT           45    // Unable to put remote file
#define ILX_ERR_NOSECT            46    // No section lists exist
#define ILX_ERR_SECTION           47    // Invalid section handle
#define ILX_ERR_INVSECT           48    // Section does not exist
#define ILX_ERR_INTABLE           49    // Unable to read system table
#define ILX_ERR_OUTABLE           50    // Unable to write system table
#define ILX_ERR_BADPARAM          51    // Invalid function parameter
#define ILX_ERR_NODATA            52    // No data to process
#define ILX_ERR_NOMAPPING         53    // No fields are mapped
#define ILX_ERR_INVREQ            54    // Invalid request
#define ILX_ERR_PASSWORD          55    // Invalid password
#define ILX_ERR_NOMOREDATES        62    // No Calendar slots
#define ILX_ERR_MAXRECORDS        63    // Exceeded allowed Cardfile records
#define ILX_ERR_BADFLDMAPSIZE     64    // No match on field map size
#define ILX_ERR_FLDMAPEENTRYOUTOFRANGE 65 // Field map fails sanity check
#define ILX_ERR_INUSE             66    // File is in use or locked
#define ILX_ERR_FILETYPE          67    // Invalid file type
#define ILX_ERR_COM_NOFILE        68    // Unable to find remote file
#define ILX_ERR_COM_NODIR        69    // Unable to find remote directory
#define ILX_ERR_COM_NOSPACE       70    // No space on remote for data
#define ILX_ERR_COM_INUSE        71    // Remote file is in use
#define ILX_ERR_COM_NORAM        72    // Unable to use RAM card
#define ILX_ERR_COM_NODISK       73    // No disk space on PC to receive
#define ILX_ERR_INREC            74    // Unable to read record from file
#define ILX_ERR_OUTREC           75    // Unable to write record to file
#define ILX_ERR_COM_SHARP        76    // Sharp Wizard data error
#define ILX_ERR_CANT_USE_ILTIF    77    // Logic/Build Error
#define ILX_ERR_INSUFF_ACCESS_RIGHTS 78 // Insufficient rights to file
#define ILX_ERR_BAD_SYNC_OPTION   79    // Bad option of cannot be done
#define ILX_ERR_CANT_DELBAK       80    // Unable to delete backup file
#define ILX_ERR_FILEFORMAT        81    // Invalid file format
#define ILX_ERR_MUSTMAP           82    // User must first map fields

/*-----
 * Always use error code #83 from within the ILERROR macro.
 *-----*/
#define ILX_ERR_INTERNAL          83    // Internal err; see ILERRORS.LOG
#define ILX_ERR_DBOPENFAILED      84    // Unable to access app. database
#define ILX_ERR_NO_DB_APP        85    // Unable to access DB application
#define ILX_ERR_NO_NEWFILE       86    // New file cannot be created
#define ILX_ERR_APIFAILED        87    // API initialization failed
#define ILX_ERR_NETFAILED        88    // Network access failed
#define ILX_ERR_NOLOGON          89    // Application logon failure
#define ILX_ERR_GETPROCADDRESS    90    // GetProcAddress failed
#define ILX_ERR_WHATFIELDS_SOURCE 91    // Source WhatFields failed
#define ILX_ERR_WHATFIELDS_TARGET 92    // Target WhatFields failed
#define ILX_ERR_DEVICE           93    // Unable to access remote device
#define ILX_ERR_DEVICE_READ      94    // Unable to read remote device
#define ILX_ERR_DEVICE_WRITE     95    // Unable to write remote device
#define ILX_ERR_SOURCE_SESSION_BEGIN 96 // SourceXlator ILBeginSession ERR
#define ILX_ERR_TARGET_SESSION_BEGIN 97 // TargetXlator ILBeginSession ERR
#define ILX_ERR_MCW_NOT_UP       98    // Magic Cap (MCW) not running
#define ILX_ERR_NO_MAGICXLATE     99    // No Magic Xlate (MCW only)
#define ILX_ERR_NO_MCW_OWNER     100   // MCW not personalized
#define ILX_ERR_TBLREADONLY      101   // System table is read-only
#define ILX_ERR_APP_VERSION      102   // can't access app; wrong version
#define ILX_ERR_INVALID_FILE     103   // Invalid file format, too few fields
#define ILX_ERR_RESYNC           104   // User wants to re-sync from scratch

//----- The resource compiler does not need to see the remainder of this header
#ifndef RC_INVOKED

//----- Special handling for C++ code
#ifdef __cplusplus
    extern "C" {
#endif // __cplusplus

/*-----
 * Limits and sizes.
 *-----*/
#define ILX_MAX_APPNAME          26    // Size of application name
#define ILX_MAX_DIR              MAX_DIR // Max size of directory name
#define ILX_MAX_DRIVER           9     // Size of driver name

```

```

#define ILX_MAX_EXT          30          // Size of file extension
#define ILX_MAX_FLDNAME      31          // Size of field name
#define ILX_MAX_FNAME       MAX_FILE_NAME // Size of file name and ext
#define ILX_MAX_PATH        MAX_PATH    // Max size of path name
#define ILX_MAX_PIMTYPES     10          // Max number of PIM types
#define ILX_MAX_PREFIX       17          // Size of field prefix
#define ILX_MAX_PSWD         25          // Size of password string
#define ILX_MAX_SECTION      26          // Size of section name
#define ILX_MAX_TYPEDESC     21          // Size of type description
#define ILX_MAX_USERDIR      32          // Max size of user dir. name

/*-----
 * System table names.
 *-----*/
#define ILX_STR_WIN16_TABLE  "tables.itb" // 16-bit table name
#define ILX_STR_WIN32_TABLE  "tbl32.itb"  // 32-bit table name

/*-----
 * Various system and section attributes.
 *-----*/
#define ILX_ATT_DEFAULT      0x00000040L // Default phone number
#define ILX_ATT_GROUPS       0x00000400L // Groups-capable application
#define ILX_ATT_TODO         0x00000200L // Todo-capable application
#define ILX_ATT_ZOOMEXT      0x00010000L // Zoomer extensions

/*-----
 * Flag bits in the Flags member of the GLOBAL DATA STRUCTURE.
 * NOTE: the 'APPEND_TO_LOGS' flag is typically left un-set by the
 * calling App so that the first translate task in a session or
 * multi-session operation causes logs to start over afresh. The engine
 * then sets 'APPEND_TO_LOGS' for subsequent translate tasks in the same
 * session or multi-session operation.
 *-----*/
#define ILX_FLAG_APPEND_TO_LOGS      0x00000001L
#define ILX_FLAG_MULTISESSION_LOGS   0x00000002L
#define ILX_FLAG_DISABLE_SST_TAGGING 0x00000004L
#define ILX_FLAG_DISABLE_SST_FILTERS 0x00000008L

/*-----
 * Set the next flag before establishing an ILX session whose sole purpose
 * is to do Field Mapping ... no translation. Translators' ILBeginSession
 * functions may do less work when this flag is set.
 *-----*/
#define ILX_FLAG_FIELD_MAPPING_ONLY  0x00000010L

/*-----
 * Set the next flag before when performing translation in "unattended" mode,
 * ie, the user is not physically present. The user may, for example, be
 * dialing in remotely and cannot respond to ANY dialogs. Therefore NO
 * dialogs or message boxes should be raised when this flag is set.
 *-----*/
#define ILX_FLAG_UNATTENDED_MODE      0x00000020L

/*-----
 * Set this flag if ILX16.EXE should not be removed during shutdown
 * processing. This flag was introduced in support of IntelliSync
 * for Pilot. It avoid starting and removing ILX16 repeatedly for
 * each individual section.
 *-----*/
#define ILX_FLAG_KEEP_ILX16_UP        0x00000040L

/*-----
 * Set this flag if the UI has already validated the file and/or database
 * name, and when the validation rules are not known to the ILX engine.
 *-----*/
#define ILX_FLAG_NO_FILE_VALIDATION    0x00000080L

/*-----
 * The ILX_FLAG_FIRST_DOTTRANSLATE flag is set in begin.c and cleared in
 * xlate.c at the conclusion of the first 'doTranslate' call in a session.
 * This controls the setting of ILTR_FLAG_FIRST_XLATE.
 *-----*/
#define ILX_FLAG_FIRST_DOTTRANSLATE    0x00000100L

/*-----

```

```

* Set this flag to cause an IMPORT operation to call the Chooser.
*-----*/
#define ILX_FLAG_IMPORT_SELECTED      0x00000200L

/*-----*/
* Set this flag to cause ILX_MapILTRErrorCode to pass on any ILTR_ERR code
* for which there is no explicit mapping to a known ILX_ERR code. This
* allows the ILX caller to take action on ILTR_ERR codes which are unknown
* to the engine and which the engine has no need to understand.
*-----*/
#define ILX_FLAG_PASS_ILTR_ERR_CODES  0x00000400L

/*-----*/
* Set this flag if the UI has already confirmed whether the user wishes
* to update the field list for a handheld device.
*-----*/
#define ILX_FLAG_NO_REMOTE_CONFIRM    0x00000800L

/*-----*/
* Enumerated types.
*-----*/

typedef enum                                // REQUEST TYPE
{
    ILX_ACT_NONE                = 0,        // No action
    ILX_ACT_TRANSLATE            = 1,        // Translate
    ILX_ACT_MERGE                = 2,        // Merge
} ILX_ACTION;

typedef enum                                // ENVIRONMENT TYPE
{
    ILX_ENV_NORMAL,              // Normal environment
    ILX_ENV_WINPAD,              // WinPad environment
    ILX_ENV_LITE,                // Lite environment
    ILX_ENV_MAGICXCHANGE,        // Magic Xchange environment
    ILX_ENV_ACHATES              // Intel Achates environment
} ILX_ENV;

typedef INT16 ILX_BOOL;                    // BOOLEAN VALUES
#define ILX_FALSE                0          // False
#define ILX_TRUE                 1          // True

typedef INT16 ILX_ACCESS;                  // ACCESS TYPES
#define ILX_ACCESS_NONE          0          // Not applicable
#define ILX_ACCESS_DDE           1          // Dynamic Data Exchange
#define ILX_ACCESS_DBASE         2          // dBASE database
#define ILX_ACCESS_FILE          3          // File access
#define ILX_ACCESS_PDX           4          // Paradox database
#define ILX_ACCESS_CDF           5          // Ascii file type
#define ILX_ACCESS_ODBC          6          // ODBC database
#define ILX_ACCESS_NEW1          7          // Unassigned
#define ILX_ACCESS_NEW2          8          // Unassigned

typedef INT16 ILX_OPTION;                  // RECONCILIATION OPTIONS
#define ILX_OPT_REPLACE          0          // Replace old item with new
#define ILX_OPT_IGNORE           1          // Ignore new item
#define ILX_OPT_NOTIFY           2          // Prompt user for resolution
#define ILX_OPT_INSERT           3          // Insert duplicate item
#define ILX_OPT_UPDATE           4          // Update existing item
#define ILX_OPT_MERGE            5          // Merge new items (Sharp)
#define ILX_OPT_ACCEPT_1         6          // Accept item from first file
#define ILX_OPT_ACCEPT_2         7          // Accept item from second file
#define ILX_OPT_NONE             8          // No option selected
#define ILX_OPT_DELETE           9          // Remove existing item
#define ILX_OPT_DELTA_ACK        10         // a FastSync Unload Action Code

typedef INT16 ILX_PIM;                    // PERSONAL INFORMATION MANAGERS
#define ILX_PIM_APPT             0x0001    // Appointment Book
#define ILX_PIM_DATA             0x0002    // Database
#define ILX_PIM_DOCUMENT         0x0004    // Word Processing Document
#define ILX_PIM_MEMO             ILX_PIM_DOCUMENT // Obsolete, changed to Document
#define ILX_PIM_PHONE            0x0008    // Phone Book
#define ILX_PIM_TODO             0x0010    // To do list
#define ILX_PIM_GROUPS           0x0020    // Phone groups
#define ILX_PIM_OUTLINE          0x0040    // Outline

```

```

#define ILX_PIM_CALL          0x0080      // Calls
#define ILX_PIM_SPREAD        0x0100      // Spreadsheet
#define ILX_PIM_EXPENSE       0x0200      // Expense
#define ILX_PIM_ALL           0xffff      // All

//----- Use ILX_RANGE values when calling ILX_SetApptRange
typedef INT16 ILX_RANGE;                // APPOINTMENT RANGE
#define ILX_RANGE_ALL         0           // All
#define ILX_RANGE_FUTURE      1           // Future only
#define ILX_RANGE_NONE        2           // None
#define ILX_RANGE_DEFAULT     3           // Use system default

typedef INT16 ILX_TERM;                 // FIELD TERMINATORS
#define ILX_TERM_COLON        ':'         // Colon
#define ILX_TERM_COMMA        ','         // Comma
#define ILX_TERM_EOS          '\0'        // End of field
#define ILX_TERM_PERIOD        '.'         // Period
#define ILX_TERM_SEMI         ';'         // Semicolon
#define ILX_TERM_SPACE        ' '         // Space
#define ILX_TERM_TAB          '\t'        // Tab
#define ILX_TERM_EOL          0xff        // End of line
#define ILX_TERM_DASH         '-'         // DASH
#define ILX_TERM_X            'x'         // x (phone extension)

typedef INT16 ILX_TYPE;                 // FIELD TYPES
#define ILX_TYPE_TEXT          'A'         // Alphanumeric field
#define ILX_TYPE_BOOL          'B'         // Boolean field
#define ILX_TYPE_DATE          'D'         // Date field
#define ILX_TYPE_NUMBER        'N'         // Number field
#define ILX_TYPE_PHONE         'P'         // Phone field
#define ILX_TYPE_TIME          'T'         // Time field
#define ILX_TYPE_BINARY        'Y'         // Binary field

//----- Currently assigned sub-section type codes in ILX_SECTION
typedef BYTE ILX_SUB_SECTION;           // SUB-SECTIONS in ILX_SECTION
#define ILX_SUBSECT_MAIN       0           // Main section, not a sub-section
#define ILX_SUBSECT_EVENT      1           // Event, anniversary, special day
#define ILX_SUBSECT_HOLIDAY    2           // Holiday, vacation day
#define ILX_SUBSECT_BUSCARD    3           // Business card
#define ILX_SUBSECT_TEL2       4           // TEL2, secondary phone book
#define ILX_SUBSECT_TEL3       5           // TEL3, tertiary phone book
#define ILX_SUBSECT_USER1      6           // USER1, user defined data
#define ILX_SUBSECT_USER2      7           // USER2, user defined data
#define ILX_SUBSECT_USER3      8           // USER3, user defined data
#define ILX_SUBSECT_CALL       9           // Call, sub-section of To do
#define ILX_SUBSECT_UNDATED    10          // Undated To do
#define ILX_SUBSECT_NESTMEMO   11          // Nested memos (real outlines)
#define ILX_SUBSECT_TELTODO    12          // ToDo in Telephone section
#define ILX_SUBSECT_BITMAP     13          // Bitmap memo
#define ILX_SUBSECT_BITMAP EVT 14          // Bitmap event
#define ILX_SUBSECT_REMIND2     15          // WinLink (only) Reminder 2
#define ILX_SUBSECT_DATABASE    16          // Unnamed subtype
#define ILX_SUBSECT_NEW_1 ILX_SUBSECT_DATABASE // OBSOLETE name, use DATABASE
#define ILX_SUBSECT_NEW_2      17          // Unnamed subtype
#define ILX_SUBSECT_NEW_3      18          // Unnamed subtype

/*-----
 * When setting the ILX_VAL_DATE_RANGE_START and ILX_VAL_DATE_RANGE_END
 * values, supply absolute date as number of days since 1900, or use
 * ILX_DATE_RANGE_UNLIMITED.
 *-----*/
#define ILX_DATE_RANGE_UNLIMITED (0)

typedef enum                            // ENGINE STATES
{
    ILX_STATE_DOWN              = 0x0000,  // Engine is not started
    ILX_STATE_UP                = 0x0001,  // Engine is started
    ILX_STATE_ACTIVE            = 0x0002   // Session is active
} ILX_STATE;

typedef enum                            // TABLE OPTION TYPES
{
    ILX_TBL_APPLOC,              // Application location
    ILX_TBL_COMPORT,            // COM port for device
    ILX_TBL_DOCNAMES,           // Use document names

```

```

    ILX_TBL_MODNAME,           // Translator module name
    ILX_TBL_RAMCARD,          // Use RAM card on device
    ILX_TBL_RECONCILE,        // Reconciliation option
    ILX_TBL_SYSTYPE,         // System type
    ILX_TBL_FILEXTS,         // File extensions
    ILX_TBL_SYSCCLASS,       // System class
    ILX_TBL_APPTRANGE,       // Default appointment range
    ILX_TBL_TODORANGE,       // Default todo range
    ILX_TBL_MAIN_SYSTEM,     // Main system
    ILX_TBL_MAIN_CLASS,      // Main class
    ILX_TBL_PRODUCT_ID,      // Product ID
    ILX_TBL_SYS_ACCESS_TYPE,  // System access type
    ILX_TBL_SYNC_CAPABLE,    // System Sync Capable?
    ILX_TBL_BUNDLED,         // Bundled product?
    ILX_TBL_SEC_FILENAME     // Filename for section
) ILX_TBLOPT;

typedef enum                  // VALUE TYPES
{
    ILX_VAL_APPTRANGE,       // Appointment range
    ILX_VAL_ASCIIINAMES,    // ASCII field names in file
    ILX_VAL_ASCIISEP,       // ASCII field separator
    ILX_VAL_COMPORT,        // COM port for device
    ILX_VAL_DATE_RANGE_START, // Use long absolute date
    ILX_VAL_DATE_RANGE_END, // Use long absolute date
    ILX_VAL_DOCNAMES,       // Use document names
    ILX_VAL_ENVIRON,        // Engine environment
    ILX_VAL_FANREPEAT,      // Fan out repeating types?
    ILX_VAL_HELPCONTEXT,    // Help context number
    ILX_VAL_HELPFILE,       // HELP file name
    ILX_VAL_HELPOPEN,       // Help context for Open File
    ILX_VAL_KEEFILES,       // Keep intermediate files
    ILX_VAL_LOGFILE,        // Create log file
    ILX_VAL_PASSWORD,       // File or user password
    ILX_VAL_RAMCARD,        // Use RAM card on device
    ILX_VAL_REMFILE,        // Force file removal
    ILX_VAL_SESSIONID,      // Session ID
    ILX_VAL_SYNCHRONIZE,    // TRUE for synchronization
    ILX_VAL_TODORANGE,      // To Do range
    ILX_VAL_VWRHELP,        // Use Viewer help system
    ILX_VAL_CBPROGRESS,     // Progress/Error Callback
    ILX_VAL_TIMERINTERVAL,  // Milliseconds
    ILX_VAL_CANCELREQUEST,  // Set to TRUE to CANCEL xlate
    ILX_VAL_TOTAL_ITEMS,    // Total items processed
    ILX_VAL_STAY_CONNECTED, // Stay connected after transfer?
    ILX_VAL_COMMHELPCONTEXT, // Help context number for Comm
    ILX_VAL_SOURCE_XTRA_DATA, // Source Translator xtra data
    ILX_VAL_TARGET_XTRA_DATA, // Target Translator xtra data
    ILX_VAL_HWINDOW,        // Change "startup" window handle
    ILX_VAL_USETABLE,       // Use values in system table
    ILX_VAL_ISCONNECTED,    // Is handheld connected?
    ILX_VAL_FLAGS,          // 32 miscellaneous flag bits
    ILX_VAL_SOURCE_WHATFIELDS, // Enable Source WhatFields?
    ILX_VAL_TARGET_WHATFIELDS, // Enable Target WhatFields?
    ILX_VAL_OKTP_THRESHOLD, // OKToProceed dialog threshold
    ILX_VAL_DIALOG_FONT     // Font to use in dialogs
} ILX_VALUE;

/*-----
 * Data types.
 *-----*/
typedef INT16 ILX_HAPP;           // APPLICATION HANDLE
typedef INT16 ILX_HSECTION;      // SECTION HANDLE
typedef UINT32 ILX_ATTR;         // ATTRIBUTE TYPE
typedef INT16 ILX_ID;            // IDENTIFIER TYPE
typedef UINT32 ILXTR_HAPPSESSION; // SESSION HANDLE

typedef struct                   // SESSION PARAMETERS
{
    char szInstallDir[ILX_MAX_DIR]; // IntelliLink directory
    char szPswd[ILX_MAX_PSWD];      // User password
    char szAppFile[ILX_MAX_PATH];   // Application file name
    ILX_HAPP hApp;                  // Application handle
    IL_HWIN hParentWnd;             // Parent window handle
    ILXTR_SYNC_OPTION nSynchronize; // Synchronize flag

```

```

    INT32 lDateRangeStart;           // Starting date range
    INT32 lDateRangeEnd;            // Ending date range
    UINT32 Flags;                   // Miscellaneous flags
    WORD hSessionID;                // Session ID
} ILXTR_INITPARMS, *ILXTR_PINITPARMS;

typedef struct                      // SYSTEM STRUCTURE
{
    ILX_HAPP hApp;                  // System handle
    char name[ILX_MAX_APPNAME];     // System name
    ILX_BOOL bIsDevice;             // Is this a device?
    ILX_BOOL bHasFiles;             // Does system have files?
    ILX_PIM capable;                // Capabilities attributes
    ILX_ATTR attribs;               // System attributes
} ILX_APP, IL_DIST *ILX_PAPP;

typedef struct                      // SECTION STRUCTURE
{
    ILX_HSECTION hSection;          // Section handle
    ILX_PIM nType;                  // Section type
    char name[ILX_MAX_APPNAME];     // Section name
    char code[ILX_MAX_SECTION];     // Section code
    char ext[ILX_MAX_EXT];          // Default file extension

    //----- The following byte has been reassigned from padChar to subSection
    #ifdef ILX_USE_OBSOLETE_PADCHAR
        char padChar;               // File name pad character
    #else
        BYTE subSection;            // Section sub-type code
    #endif

    ILX_ATTR attribs;               // Section attributes
} ILX_SECTION, IL_DIST *ILX_PSECTION;

typedef struct                      // FIELD STRUCTURE
{
    char name[ILX_MAX_FLDNAME];     // User visible name
    INT16 itemNo;                   // Item number within field
    INT16 mapIndex;                 // Index to mapped field
    char label[ILX_MAX_FLDNAME];    // Internal field label
    ILX_TYPE type;                  // Field type
    INT32 width;                    // Maximum length
    ILX_TERM delimit;               // Terminating delimiter
    char prefix[ILX_MAX_PREFIX];    // Optional field prefix
    char typeDesc[ILX_MAX_TYPEDESC]; // Optional type description
    INT16 order;                    // Sort order of field
    INT16 assoc;                    // Index of associated field
    ILX_ATTR attribs;               // Field attributes
} ILX_FIELD, IL_DIST *ILX_PFIELD;

typedef struct                      // FIELD MAP LIST
{
    ILX_ID nMapId;                  // Field map ID
    char szMapName[ILX_MAX_FLDNAME]; // Field map name
    ILX_BOOL bIsDefault;            // Is this default map?
} ILX_MAPLIST, IL_DIST *ILX_PMAPLIST;

//----- Translator module HANDLE data type
typedef struct                      // Translator HANDLE
{
    IL_HINST hXlator;               // Xlator or engine instance
    int nInternalID;                // Translator ID (must be zero)
} IL_HXLATOR, IL_DIST *IL_PHXLATOR;

/*-----
 * ILX Globals -- Note that there are now TWO structures which make up the
 * ILX global data, the "standard" globals and the "extended" globals. The
 * extended globals structure is allocated by ILX_Startup and freed by
 * ILX_Shutdown. The extended area is reached as follows:
 *
 *          (_ilx_globals->pExt->nDataValue)
 *
 * However, as with all references to ILX globals, access to the extended
 * globals should always use the access macros (eg, ILXGL_nDataValue).

```



```

-----*/
//----- Available slack bytes in standard global structure for new fields
#define ILX_MAX_FREE 26

//----- Available slack bytes in extended global structure for new fields
#define ILX_MAX_FREE_EXT 256

//----- ILX EXTENDED GLOBALS STRUCTURE
typedef struct // EXTENDED GLOBAL STRUCTURE
{
    char szUserDir [ILX_MAX_USERDIR]; // User (directory) name

    /*-----
    * If you extend this structure, count the bytes, and adjust (down!)
    * ILX_MAX_FREE_EXT above to keep size consistent with previous releases.
    *-----*/
    char FreeSpace [ILX_MAX_FREE_EXT]; // Free space
} ILX_EXTG, IL_DIST* ILX_PEXTG;

//----- ILX STANDARD GLOBALS STRUCTURE
typedef struct // STANDARD GLOBAL STRUCTURE
{
    /*-----
    * Engine settings and general options.
    *-----*/
    int wrc; // Translator return code
    ILX_ACTION action; // Action requested
    ILX_BOOL bFanRepeat; // Fan out recurring items?
    ILX_BOOL bFirstXlate; // First translation?
    ILX_BOOL keepFiles; // Save intermediate files?
    ILX_BOOL log; // Create log file?
    ILX_BOOL remFile; // Remove file before import?
    ILX_ENV nEnviron; // Engine environment
    ILX_RANGE apptRange; // Appointment range
    ILX_RANGE todoRange; // Todo range
    ILX_STATE engineState; // Engine state

    /*-----
    * Directories and files.
    *-----*/
    int nCurDrive; // Current drive
    char szCurDir[ILX_MAX_DIR]; // Current working directory
    char szDir[ILX_MAX_DIR]; // IntelliLink working directory
    char szPswd[ILX_MAX_PSWD]; // File/user password
    char szSourceFile1[ILX_MAX_PATH]; // Source file name 1
    char szSourceFile2[ILX_MAX_PATH]; // Source file name 2
    char szTargetFile[ILX_MAX_PATH]; // Target file name
    IL_HANDLE hTable; // Handle to table

    /*-----
    * Systems and sections.
    *-----*/
    ILX_HAPP hSourceApp; // Handle to source system
    ILX_HAPP hTargetApp; // Handle to target system
    ILX_HSECTION hSourceSect; // Handle to source section
    ILX_HSECTION hTargetSect; // Handle to target section
    IL_HANDLE hSourceAppRec; // Handle to source system
    IL_HANDLE hSourceSectRec; // Handle to source section
    IL_HANDLE hTargetAppRec; // Handle to target system
    IL_HANDLE hTargetSectRec; // Handle to target section

    /*-----
    * Application-specific settings from initialization file.
    *-----*/
    char AsciiSep; // Ascii field separator
    ILX_BOOL AsciiNames; // Names as first Ascii record?

    /*-----
    * Help information.
    *-----*/
    UINT32 nHelpContext; // Help context number
    UINT32 nHelpOpen; // Help context for File Open

```

```

char szHelpFile(ILX_MAX_FNAME);          // HELP file name
ILX_BOOL bVWRHelp;                      // Use VWR help?

/*-----
 * Communication settings.
 *-----*/
int nFileList; -                        // Number of files in list
ILX_BOOL bConnected;                   // Is device connected?
ILX_BOOL bDocument;                    // Use Document names?
ILX_BOOL bRamCard;                     // Use RAM card on device?
IL_HINST hCommDLL;                     // Communication DLL
IL_HINST hEngineDLL;                   // Handle to engine DLL
IL_HINST hOtherSrcDLL;                 // Handle to other source DLL
IL_HINST hOtherTarDLL;                 // Handle to other target DLL
IL_HANDLE hFileList;                   // Handle to file list
WORD nComPort;                         // Communication port

/*-----
 * Windows and handles.
 *-----*/
IL_HINST hInstance;                    // Instance handle
IL_HWIN hLineWin;                      // Line drawing window handle
IL_HWIN hMapWin;                       // Field mapping window handle
IL_HWIN hWindow;                       // Parent window handle
WORD hSessionID;                       // Session ID

/*-----
 * Synchronization settings.
 * To specify unlimited date range, use ILX_DATE_RANGE_UNLIMITED (-1)
 * for both Start and End.
 *-----*/
INT32 lDateRangeStart;                  // Earliest appointment date
INT32 lDateRangeEnd;                   // Latest appointment date
ILXTR_SYNC_OPTION nSynchronize;         // Short int; see ilxtr.h

/*-----
 * System error code (non-ILX errors).
 *-----*/
int nXlatorError;                      // Translator error code (ILTR)
int nSystemError;                      // Application error code

/*-----
 * Optional application-specified callback for Progress & Error reporting
 *-----*/
IL_CBPROGRESS cbProgress;               // Progress-reporting callback
UINT uTimerInterval;                   // In milliseconds (ZERO = 55)
ILX_BOOL bCancelRequest;                // TRUE to request cancelation
INT16 nXlateDataErrs;                  // Number of Data errors
IL_HWIN hProgWin;                      // Handle for Progress Window

/*-----
 * Count of records processed.
 *-----*/
INT32 lTotalRecords;                   // Total records processed
ILX_BOOL bStayConnected;                // Stay connected after transfer?

//----- Communications prompt help context number
UINT32 nCommHelpContext;                // Help context number

//----- Type of handheld connected to
INT16 nConnectedType;                   // Handheld type

/*-----
 * Handles for current (internal or external) source/target translators.
 * Added 3/31/95 by Bob -- ILX_MAX_FREE adjusted from 68 to 64 (16-bit).
 *-----*/
IL_HXLATOR hSourceDLL;                  // Xlator handle of source DLL
IL_HXLATOR hTargetDLL;                  // Xlator handle of target DLL

/*-----
 * Pointers intended to be used to pass any additional translator specific
 * data from the UserInterface to a specific translator. The fields are
 * typed as void pointers to indicate and allow them to point to any
 * data the UI and translator need.
 * Added 4/19/96 by DaveMc -- ILX_MAX_FREE adjusted from 64 to 56.

```



```

/*-----*/
void * pSourceXtraData;          // Pointer to extra source data
void * pTargetXtraData;         // Pointer to extra target data

/*-----*/
/* Flag bUseTable causes option settings to be taken directly from
 * the system table, whenever possible. This means that options such
 * as the reconciliation, RAM card, and document options (for example)
 * will be taken from the system record rather than the ILX setting.
 * The default value of this option is set to ILX_TRUE.
 *-----*/
ILX_BOOL bUseTable;              // Take settings from table?

/*-----*/
/* Miscellaneous engine flags word, use ILX_FLAG_xxxx #defines.
 * Added 1/24/96 by DavidB -- ILX_MAX_FREE adjusted from 54 to 50.
 *-----*/
UINT32 Flags;                    // Misc. Flag Bits

/*-----*/
/* Handles to application session returned from call to callback
 * function ILBeginSession. Reduced ILX_MAX_FREE to 42 bytes.
 *-----*/
ILXTR_HAPPSESSION hSrcAppSession; // Source session handle
ILXTR_HAPPSESSION hTarAppSession; // Target session handle

/*-----*/
/* Boolean settings to drive whether or not field lists should be
 * dynamically updated. ILX_MAX_FREE to 38 bytes
 *-----*/
ILX_BOOL bSrcWhatFields;         // Source system whatfields
ILX_BOOL bTarWhatFields;         // Target system whatfields

/*-----*/
/* Pointer to callback function used to ask user's permission before
 * trampling over his or her data. ILX_MAX_FREE to 34 bytes
 *-----*/
UINT32 OKTP_Threshold;           // OKToProceed dialog threshold

/*-----*/
/* Font to be used drawing controls in dialogs. ILX_MAX_FREE to 32 bytes
 *-----*/
HFONT hDialogFont;

/*-----*/
/* Handle/pointer to ILX "Extended" globals. ILX_MAX_FREE to 26 bytes
 *-----*/
IL_HANDLE hExt;                  // Handle for ILX ext. globals
ILX_PEXTG pExt;                  // Pointer to ILX ext. globals

/*-----*/
/* ILX header version number. ILX_MAX_FREE to 24 bytes
 *-----*/
INT16 version;                   // set to ILX_CURRENT_VERSION

/*-----*/
/* If you extend this structure, count the bytes, and adjust (down!)
 * ILX_MAX_FREE above to keep size consistent with previous releases.
 *-----*/
char szFiller[ILX_MAX_FREE];     // Free space for expansion

} ILX_GLOBALS, IL_DIST *ILX_PGLOBALS;

/*-----*/
/* Function prototypes.
 * When running under MS-Windows, the global data space must be
 * allocated within the calling application to ensure reentrancy.
 * This is accomplished by declaring and passing a pointer to the
 * global data space to each function. The additional parameter
 * passed to the DLL version of ILX is hidden in macro definitions
 * supplied for each function. The actual DLL function names are
 * prefixed with 'd' to distinguish them from the macros.
 *-----*/
/*-----*/

```

```

* HIGH level API functions.
* These functions are called to initiate IntelliLink/Lite. These can
* be used in conjunction with Low-level functions. Note that none of
* the High-level functions require access to the global data structure
* nor access the IntelliLink engine DLL.
*-----*/
int IL_DECL ILX_DoExport // Initiate Export operation
( IL_HWIN hParentWnd, // Handle to parent window
  IL_PSTR pEngineDir, // IntelliLink directory
  ILX_HAPP hSourceApp, // Handle to source system
  ILX_HAPP hTargetApp, // Handle to target system
  ILX_PIM nTypes ); // Section types to include
int IL_DECL ILX_DoImport // Initiate Import operation
( IL_HWIN hParentWnd, // Handle to parent window
  IL_PSTR pEngineDir, // IntelliLink directory
  ILX_HAPP hSourceApp, // Handle to main system
  ILX_HAPP hTargetApp, // Handle to selected system
  ILX_PIM nTypes ); // Section types to include
int IL_DECL ILX_DoSynch // Initiate Synch operation
( IL_HWIN hParentWnd, // Handle to parent window
  IL_PSTR pEngineDir, // IntelliLink directory
  ILX_HAPP hSourceApp, // Handle to main system
  ILX_HAPP hTargetApp, // Handle to selected system
  ILX_PIM nTypes ); // Section types to include
ILX_HAPP IL_DECL ILX_GetAppHandle // Get application handle
( IL_PSTR pEngineDir, // IntelliLink directory
  IL_PSTR pAppName ); // Pointer to system name
ILX_HAPP IL_DECL ILX_GetAppHandleEx // Get application handle
( IL_PSTR pTableName, // System table name
  IL_PSTR pEngineDir, // IntelliLink directory
  IL_PSTR pAppName ); // Pointer to system name
int IL_DECL ILX_GetAppList // Get qualified system list
( IL_HANDLE phTable, // Pointer to table handle or NULL
  IL_PSTR pEngineDir, // IntelliLink directory
  ILX_PIM nTypes, // Section types to include
  ILX_HAPP hExcludeApp, // Handle to excluded system
  IL_HANDLE IL_DIST *hAppList, // Handle to returned system list
  ILPINT nAppList ); // Number of systems in list
int IL_DECL ILX_GetAppListEx // Get qualified system list
( IL_PSTR pTableName, // System table name
  IL_HANDLE phTable, // Pointer to table handle or NULL
  IL_PSTR pEngineDir, // IntelliLink directory
  ILX_PIM nTypes, // Section types to include
  ILX_HAPP hExcludeApp, // Handle to excluded system
  IL_HANDLE IL_DIST *hAppList, // Handle to returned system list
  ILPINT nAppList ); // Number of systems in list
int IL_DECL ILX_UsedllVersion // Use DLL or EXE version?
( ILX_BOOL bUsedll ); // Use DLL version?

/*-----*/
* LOW level API functions.
* These functions are used when the caller provides a full user
* interface component to drive translation. All Low-level functions
* require access to the global data structure and reference the
* IntelliLink engine DLL.
*-----*/
int IL_DECL ILX_dAddSection // Add new system section
( ILX_PGLOBALS _ilx_globals, // Pointer to global data
  ILX_HAPP hApp, // Handle to system
  IL_PSTR pSectionName, // Pointer to section name
  ILX_PIM nType, // Section type
  ILX_HSECTION *phSection ); // Returned section handle
int IL_DECL ILX_dBeginSession // Initiate communication session
( ILX_PGLOBALS _ilx_globals ); // Pointer to global data
int IL_DECL ILX_dEndSession // Terminate communication session
( ILX_PGLOBALS _ilx_globals ); // Pointer to global data
int IL_DECL ILX_dCallGetPassword // Call translator ILGetPassword
( ILX_PGLOBALS _ilx_globals, // Pointer to ilx globals
  IL_PSTR pszTransName, // Pointer to translator name
  IL_PSTR pszAppFile, // Pointer to app. file name
  IL_PSTR pszSectName, // Pointer to app. section name
  IL_PSTR pszPassword ); // Pointer to password buffer
int IL_DECL ILX_dFlipApps // Exchange Source/Target handles
( ILX_PGLOBALS _ilx_globals ); // Pointer to global data
int IL_DECL ILX_dFullBackup // Call thru to COMM for full backup

```

```

    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      INT16 nSysClass,
      int nComPort );
int IL_DECL ILX_dFullRestore
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      int nComPort );
int IL_DECL ILX_dGetErrorText
    ( ILX_PGLOBALS _ilx_globals,
      int nErrorCode,
      IL_PSTR pText,
      int nText );
int IL_DECL ILX_dGetFieldMap
    ( ILX_PGLOBALS _ilx_globals,
      IL_HANDLE IL_DIST *hSrcList,
      IL_HANDLE IL_DIST *hTarList,
      ILPINT nSrcList,
      ILPINT nTarList,
      int nType );
int IL_DECL ILX_dGetFileName
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_HSECTION hSection,
      int nIndex,
      IL_PSTR pFileName,
      int nFileNameLen );
int IL_DECL ILX_dGetSectionList
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_PIM nTypes,
      IL_HANDLE IL_DIST *hSectList,
      ILPINT nSectList );
int IL_DECL ILX_dGetValue
    ( ILX_PGLOBALS _ilx_globals,
      ILX_VALUE nType,
      long *pIValue );
int IL_DECL ILX_dMerge
    ( ILX_PGLOBALS _ilx_globals,
      ILX_OPTION nReconcile,
      IL_PSTR pLogFile,
      ILX_BOOL bMapFields );
int IL_DECL ILX_dPutFieldMap
    ( ILX_PGLOBALS _ilx_globals,
      IL_HANDLE hSrcList,
      IL_HANDLE hTarList,
      int nSrcList,
      int nTarList );
int IL_DECL ILX_dReadTable
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_TBLOPT nOption,
      long *lValue );
int IL_DECL ILX_dRemoveSection
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_HSECTION hSection );
ILX_HAPP IL_DECL ILX_dRemoveSystem
    ( ILX_PGLOBALS _ilx_globals,
      IL_PCSTR pEngineDir,
      ILX_HAPP hSystem,
      ILX_BOOL bDeleteXlator );
int IL_DECL ILX_dRenameSection
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_HSECTION hSection,
      IL_PSTR pSectionName );
ILX_PIM IL_DECL ILX_dSectionToPIM
    ( ILX_PGLOBALS _ilx_globals,
      int nSectionType );
int IL_DECL ILX_dSelectApps
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hSrcApp,
      ILX_HAPP hTarApp );
int IL_DECL ILX_dSelectFiles
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      INT16 nSysClass,
      int nComPort );
// Pointer to global data
// Handheld system to backup
// System class of handheld
// Comport to use for I/O
// Call thru to COMM for full restore
// Pointer to global data
// Handheld system to restore
// Comport to use for I/O
// Return error text
// Pointer to global data
// ILX error code
// Pointer to returned text
// Size of text buffer
// Get field lists and map
// Pointer to global data
// Pointer to source list handle
// Pointer to target list handle
// Pointer to source field count
// Pointer to target field count
// Request type
// Get last data file name
// Pointer to global data
// System handle
// Section handle
// Index of file in list
// Pointer to returned file name
// Size of file name buffer
// Get list of sections
// Pointer to global data
// System handle
// Section types to include
// Pointer to section list handle
// Pointer to number of sections
// Get miscellaneous options
// Pointer to global data
// Option type
// Pointer to option value
// Merge two files into one
// Pointer to global data
// Reconciliation option
// Pointer to log file name
// Show field map dialog?
// Replace field list and map
// Pointer to global data
// Handle to source field list
// Handle to target field list
// Number of source fields
// Number of target fields
// Read system table
// Pointer to global data
// Application handle
// Option type
// Returned option value
// Delete system section
// Pointer to global data
// System handle
// Section handle
// Remove system from tables
// Pointer to global data
// IntelliLink directory
// System ID to be removed
// Delete xlator file also?
// Rename system section
// Pointer to global data
// System handle
// Section handle
// New section name
// Convert section to PIM type
// Pointer to global data
// Section type to convert
// Select source & target systems
// Pointer to global data
// Handle to source system
// Handle to target system
// Set source & target file names

```

```

        ( ILX_PGLOBALS _ilx_globals,
          IL_PSTR pSrcFile1,
          IL_PSTR pSrcFile2,
          IL_PSTR pTarFile );
int IL_DECL ILX_dSelectSections
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HSECTION hSrcSection,
      ILX_HSECTION hTarSection );
int IL_DECL ILX_dSetFileName
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_HSECTION hSection,
      int nIndex,
      IL_PSTR pFileName );
int IL_DECL ILX_dSetValue
    ( ILX_PGLOBALS _ilx_globals,
      ILX_VALUE nType,
      long lValue );
int IL_DECL ILX_dShowFieldMap
    ( ILX_PGLOBALS _ilx_globals );
int IL_DECL ILX_dShutDown
    ( ILX_PGLOBALS _ilx_globals );
int IL_DECL ILX_dStartup
    ( ILX_PGLOBALS _ilx_globals,
      IL_PSTR pDir,
      IL_HWIN hParentWnd );
int IL_DECL ILX_dStartupEx
    ( ILX_PGLOBALS _ilx_globals,
      IL_PSTR pDir,
      IL_HWIN hParentWnd,
      IL_PSTR pszUserID );
int IL_DECL ILX_dTranslate
    ( ILX_PGLOBALS _ilx_globals,
      ILX_OPTION nReconcile,
      IL_PSTR pLogFile,
      ILX_BOOL nMapFields );
INT16 IL_DECL ILX_dTranslateDataErrors
    ( ILX_PGLOBALS _ilx_globals );
int IL_DECL ILX_dUpdateTable
    ( ILX_PGLOBALS _ilx_globals,
      ILX_HAPP hApp,
      ILX_TBLOPT nOption,
      long lValue );

// Pointer to global data
// Source file name
// Second source file name (Merge)
// Target file name
// Select system sections
// Pointer to global data
// Handle to source section
// Handle to target section
// Get last data file name
// Pointer to global data
// System handle
// Section handle
// Index of file in list
// Pointer to returned file name
// Set miscellaneous options
// Pointer to global data
// Option type
// Option value
// Display field mapping dialog
// Pointer to global data
// Shut down translation engine
// Pointer to global data
// Start up translation engine
// Pointer to global data
// Engine directory or NULL
// Handle to parent window
// ILX_dStartup extended version
// Pointer to global data
// Engine directory pointer
// Handle to parent window
// User ID string or NULL
// Translate source to target
// Pointer to global data
// Reconciliation option
// Pointer to log file name
// Show field mapping dialog?
// Data errors during last xlate?
// Pointer to global data
// Update system table
// Pointer to global data
// Application handle
// Option type
// Option value

/*-----
 * Extended API functions (added 4/15/95). These functions provided
 * support for creating and deleting custom field maps. They also
 * provide support for saving and loading field map contents from
 * arbitrary disk files.
 *-----*/
ILX_ID IL_DECL ILX_dGetActiveFieldMap
    ( ILX_PGLOBALS _ilx_globals );
int IL_DECL ILX_dGetFieldMapEx
    ( ILX_PGLOBALS _ilx_globals,
      ILX_ID nMapId,
      IL_HANDLE IL_DIST *phSrcFlds,
      IL_HANDLE IL_DIST *phTarFlds,
      ILPINT pnSrcCount,
      ILPINT pnTarCount,
      int nQualify );
int IL_DECL ILX_dGetFieldMapList
    ( ILX_PGLOBALS _ilx_globals,
      IL_HANDLE IL_DIST *phMapList,
      ILPINT pnMapCount );
int IL_DECL ILX_dLoadFieldMapFile
    ( ILX_PGLOBALS _ilx_globals,
      int nType,
      IL_PSTR pszFileName,
      IL_PSTR pszMapName,
      ILX_ID *pnMapId );
int IL_DECL ILX_dMergeEx
    ( ILX_PGLOBALS _ilx_globals,
      ILX_ID nMapId,
      ILX_OPTION nReconcile,
      IL_PSTR pLogFile );
// Get active field map
// Pointer to global data
// Get field map (extended)
// Pointer to global data
// Field map ID to get
// Pointer to source list handle
// Pointer to target list handle
// Pointer to source field count
// Pointer to target field count
// Request qualifier
// Get list of field maps
// Pointer to global data
// Pointer to list handle
// Pointer to list item count
// Load field map from file
// Pointer to global data
// Section type to load
// Pointer to file name
// Name of new field map
// Pointer to field map ID
// Merge two files into one
// Pointer to global data
// Field map ID to use
// Reconciliation option
// Pointer to log file name

```

```

        ILX_BOOL bMapFields );           // Show field map dialog?
ILX_ID IL_DECL ILX_dNewFieldMap          // Create new field map
( ILX_PGLOBALS _ilx_globals,           // Pointer to global data
  ILX_ID nBaseMapId,                   // ID of base field map
  IL_PSTR pszMapName );                // Pointer to field map name
int IL_DECL ILX_dPutFieldMapEx           // Write field map (extended)
( ILX_PGLOBALS _ilx_globals,           // Pointer to global data
  ILX_ID nMapId,                       // Field map ID to save
  IL_HANDLE hSrcFlds,                  // Handle to source fields
  IL_HANDLE hTarFlds,                  // Handle to target fields
  int nSrcCount,                       // Number of source fields
  int nTarCount );                     // Number of target fields
ILX_BOOL IL_DECL ILX_dRemoveFieldMap     // Remove field map
( ILX_PGLOBALS _ilx_globals,           // Pointer to global data
  ILX_ID nMapId );                     // Field map ID to remove
ILX_BOOL IL_DECL ILX_dSaveFieldMapFile   // Save field map to file
( ILX_PGLOBALS _ilx_globals,           // Pointer to global data
  ILX_ID nMapId,                       // Field map ID to save
  IL_PSTR pszFileName );               // Pointer to file name
ILX_BOOL IL_DECL ILX_dSelectFieldMap     // Set active field map
( ILX_PGLOBALS _ilx_globals,           // Pointer to global data
  ILX_ID nMapId,                       // Field map ID to activate
  ILX_BOOL bSet );                     // Set or clear flag?
int IL_DECL ILX_dTranslateEx             // Translate source to target
( ILX_PGLOBALS _ilx_globals,           // Pointer to global data
  ILX_ID nMapId,                       // Field map ID to use
  ILX_OPTION nReconcile,                // Reconciliation option
  IL_PSTR pLogFile,                     // Pointer to log file name
  ILX_BOOL nMapFields );               // Show field mapping dialog?

/*-----
 * Macro to declare DLL global data in application data space.
 *-----*/
#define ILX_DECL_GLOBALS                ILX_GLOBALS _ilx_globals

/*-----
 * Macros to access error codes.
 *-----*/
#define ILX_GetLastXlatorError() (_ilx_globals.nXlatorError)
#define ILX_GetLastSystemError() (_ilx_globals.nSystemError)

/*-----
 * Macros for DLL function calls.
 *-----*/
#define ILX_AddSection(a,b,c,d) \
  ILX_dAddSection(&_ilx_globals,a,b,c,d)
#define ILX_BeginSession() \
  ILX_dBeginSession(&_ilx_globals)
#define ILX_CallGetPassword(a,b,c,d) \
  ILX_dCallGetPassword(&_ilx_globals,a,b,c,d)
#define ILX_EndSession() \
  ILX_dEndSession(&_ilx_globals)
#define ILX_FlipApps() \
  ILX_dFlipApps(&_ilx_globals)
#define ILX_FullBackup(a,b,c) \
  ILX_dFullBackup(&_ilx_globals,a,b,c)
#define ILX_FullRestore(a,b) \
  ILX_dFullRestore(&_ilx_globals,a,b)
#define ILX_GetErrorText(a,b,c) \
  ILX_dGetErrorText(&_ilx_globals,a,b,c)
#define ILX_GetFieldMap(a,b,c,d,e) \
  ILX_dGetFieldMap(&_ilx_globals,a,b,c,d,e)
#define ILX_GetFileName(a,b,c,d,e) \
  ILX_dGetFileName(&_ilx_globals,a,b,c,d,e)
#define ILX_GetSectionList(a,b,c,d) \
  ILX_dGetSectionList(&_ilx_globals,a,b,c,d)
#define ILX_GetValue(a,b) \
  ILX_dGetValue(&_ilx_globals,a,b)
#define ILX_Merge(a,b,c) \
  ILX_dMerge(&_ilx_globals,a,b,c)
#define ILX_PutFieldMap(a,b,c,d) \
  ILX_dPutFieldMap(&_ilx_globals,a,b,c,d)
#define ILX_ReadTable(a,b,c) \
  ILX_dReadTable(&_ilx_globals,a,b,c)
#define ILX_RemoveSection(a,b) \

```

```

    ILX_dRemoveSection(&_ilx_globals,a,b)
#define ILX_RemoveSystem(a,b,c) \
    ILX_dRemoveSystem(&_ilx_globals,a,b,c)
#define ILX_RenameSection(a,b,c) \
    ILX_dRenameSection(&_ilx_globals,a,b,c)
#define ILX_SectionToPIM(a) \
    ILX_dSectionToPIM (&_ilx_globals,a)
#define ILX_SelectApps(a,b) \
    ILX_dSelectApps(&_ilx_globals,a,b)
#define ILX_SelectFiles(a,b,c) \
    ILX_dSelectFiles(&_ilx_globals,a,b,c)
#define ILX_SelectSections(a,b) \
    ILX_dSelectSections(&_ilx_globals,a,b)
#define ILX_SetFileName(a,b,c,d) \
    ILX_dSetFileName(&_ilx_globals,a,b,c,d)
#define ILX_SetValue(a,b) \
    ILX_dSetValue(&_ilx_globals,a,b)
#define ILX_ShowFieldMap() \
    ILX_dShowFieldMap(&_ilx_globals)
#define ILX_ShutDown() \
    ILX_dShutDown(&_ilx_globals)
#define ILX_StartUp(a,b) \
    ILX_dStartUp(&_ilx_globals,a,b)
#define ILX_StartUpEx(a,b,c) \
    ILX_dStartUpEx(&_ilx_globals,a,b,c)
#define ILX_Translate(a,b,c) \
    ILX_dTranslate(&_ilx_globals,a,b,c)
#define ILX_TranslateDataErrors() \
    ILX_dTranslateDataErrors(&_ilx_globals)
#define ILX_UpdateTable(a,b,c) \
    ILX_dUpdateTable(&_ilx_globals,a,b,c)

/*-----
 * Macros for extended functions (added 4/15/95)
 *-----*/
#define ILX_GetActiveFieldMap() \
    ILX_dGetActiveFieldMap(&_ilx_globals)
#define ILX_GetFieldMapEx(a,b,c,d,e,f) \
    ILX_dGetFieldMapEx(&_ilx_globals,a,b,c,d,e,f)
#define ILX_GetFieldMapList(a,b) \
    ILX_dGetFieldMapList(&_ilx_globals,a,b)
#define ILX_LoadFieldMapFile(a,b,c,d) \
    ILX_dLoadFieldMapFile(&_ilx_globals,a,b,c,d)
#define ILX_MergeEx(a,b,c,d) \
    ILX_dMergeEx(&_ilx_globals,a,b,c,d)
#define ILX_NewFieldMap(a,b) \
    ILX_dNewFieldMap(&_ilx_globals,a,b)
#define ILX_PutFieldMapEx(a,b,c,d,e) \
    ILX_dPutFieldMapEx(&_ilx_globals,a,b,c,d,e)
#define ILX_RemoveFieldMap(a) \
    ILX_dRemoveFieldMap(&_ilx_globals,a)
#define ILX_SaveFieldMapFile(a,b) \
    ILX_dSaveFieldMapFile(&_ilx_globals,a,b)
#define ILX_SelectFieldMap(a,b) \
    ILX_dSelectFieldMap(&_ilx_globals,a,b)
#define ILX_TranslateEx(a,b,c,d) \
    ILX_dTranslateEx(&_ilx_globals,a,b,c,d)

//----- Special handling for C++ code
#ifdef __cplusplus
}
#endif // __cplusplus

```



```

#if !defined(__ILXAPI)

/*-----
 * Name:      ILXAPI.H
 * Purpose: Internal header file for translator API
 * Author: Copyright (c) IntelliLink, 1992
 *-----*/
#define __ILXAPI                // Signal header inclusion

#ifdef ILWIN
    #include <windows.h>           // Windows header
    #include <windowsx.h>         // Windows 16/32 compatibility
    #include "ilwin32.h"          // Compatibility macros
    #include "ilmacro.h"
    #include "ildde.h"
#endif // ILWIN

//----- Special handling for C++ code
#ifdef __cplusplus
    extern "C" {
#endif // __cplusplus

#include <time.h>                 // DOS time header
#include "litehlp.h"             // Lite context numbers

/*-----
 * Include related header files.
 *-----*/
#include "ilxdlg.h"
#include "iltr.h"
#include "ilxerr.h"
#include "litemsg.h"
#include "ilsysids.h"

/*-----
 * External variables.
 *-----*/
extern IL_HINST hDLLInstance;    // DLL instance
extern ILX_BOOL _bUseLiteDll;   // Use DLL version of UI?

/*-----
 * Data types.
 *-----*/

//----- I/O Direction
typedef ILTR_DIRECTION          ILX_IO;    // Direction type
#define ILX_IO_IMPORT           ILTR_IMPORT // Import
#define ILX_IO_EXPORT           ILTR_EXPORT // Export
#define ILX_IO_BOTH             ILTR_EXPORT+1 // Import/Export
#define ILX_IO_SYNC             ILTR_EXPORT+2 // Synchronize

//----- Values for 'nPutWhere' member of File List
#define ILX_PUT_NOWHERE 0        // file is NOT for output
#define ILX_PUT_TO_SOURCE 1      // file is for output To "Source" (sync only!)
#define ILX_PUT_TO_TARGET 2      // file is for output To Target

typedef struct                  // File list
{
    ILX_BOOL bContainsData;    // Does file contain data?
    ILX_BOOL bRamCard;         // State of RAM card
    INT16 nPutWhere;           // use ILX_PUT_XXX #defines
    char szRealName[MAX_PATH]; // Original file name
    char szTempName[MAX_PATH]; // Temporary file name
} ILX_FILIST, IL_DIST *ILX_PFILIST;

typedef struct                  // Record pointers
{
    ILTB_PSECREC pSrcSec;      // Pointer to source section
    ILTB_PSECREC pTarSec;      // Pointer to target section
    ILTB_PSYSREC pSrcApp;      // Pointer to source system
    ILTB_PSYSREC pTarApp;      // Pointer to target system
} ILX_RECS, IL_DIST *ILX_PRECS;

typedef struct                  // Header for map files
{

```



```

    INT16 nSignature;           // File signature
    INT16 nMajorVersion;       // Major version number
    INT16 nMinorVersion;       // Minor version number
    INT16 nMapLen;             // Length of map record
    INT16 nConLen;             // Length of config record
    INT16 nSrcLen;             // Length of source field list
    INT16 nTarLen;             // Length of target field list
    INT32 nUnused;             // Reserved for future use
} ILX_MAPHDR, IL_DIST *ILX_PMAPHDR;

/*-----
 * Constants.
 *-----*/
#define ILX_BEGINCB           "ILBeginSession" // Begin session callback
#define ILX_COMM_DLL          "ilecomm.dll"    // Communication DLL name
#define ILX_ENDCB             "IEndSession"    // End session callback
#define ILX_ENGINE_SECTION    "ilxeng"        // Section in WIN.INI
#define ILX_EXPORT_DRIVER     "IExport"        // Export driver
#define ILX_EXPORTWP_DRIVER   "IExportWP"      // Export WP driver
#define ILX_TABLES_FILE       "tables.itb"     // ILTB tables file name
#define ILX_FIL_WILDCARD      "ILX*.FIL"       // Wild card for filter files
#define ILX_HELP_PROP         "HelpNo"        // Window property for Help
#define ILX_HELP_FILE         "ilxlate.hlp"    // Default Windows Help file
#define ILX_IMPORT_DRIVER     "IImport"        // Import driver
#define ILX_IMPORTWP_DRIVER   "IImportWP"      // Import WP driver
#define ILX_LITE_EXEC         "ilxlate.exe"    // IntelliLink/Lite executable
#define ILX_MAX_FRAG          25               // File fragmentation ratio
/*-----
 * MapField field index values for unmapped fields
 *-----*/
#define ILX_UNMAPPED          ILTR_UNMAPPED    // Unmapped field
#define ILX_UNMAPPED_BUT_TAGGED ILTR_UNMAPPED_BUT_TAGGED // special value

#define ILX_WHATFIELDS        "ILWhatFields"   // Fields identification function
#define ILX_WHATFIELDS_EX     "ILWhatFieldsEx" // Extended version of WhatFields
#define ILX_GETPASSWORD       "ILXGetPassword" // Translator get password proc.
#ifdef WIN32
    #define ILX_ENGINE_DLL     "ilx32.dll"     // 32-Bit Engine DLL
    #define ILX_LITE_DLL       "illite32.dll"   // 32-Bit UI DLL
#else
    #define ILX_ENGINE_DLL     "ilxw.dll"      // 16-Bit Engine DLL
    #define ILX_LITE_DLL       "ilxlate.dll"   // 16-Bit UI DLL
#endif // WIN32

//----- Map file constants
#define ILX_MAP_SIGNATURE     0x0C0E           // Map file signature
#define ILX_MAP_MAJOR_VER     1                // Major version number
#define ILX_MAP_MINOR_VER     0                // Minor version number

//----- "DLL" function codes for translator code resources on the Macintosh
#define ILX_CALL_EXPORT        100             // IExport
#define ILX_CALL_IMPORT        101             // IImport
#define ILX_CALL_WHAT_FIELDS   102             // ILWhatFields
#define ILX_CALL_WHAT_FIELDS_EX 103            // ILWhatFields extended version
#define ILX_CALL_GET_PASSWORD  104             // ILGetPassWord
#define ILX_CALL_BEGIN_SESSION 105             // ILBeginSession
#define ILX_CALL_END_SESSION   106             // IEndSession

/*-----
 * Messages.
 *-----*/
#define ILX32_FINI_TRANS       WM_USER + 900

/*-----
 * Default Help context numbers.
 *-----*/
#define ILX_HELP_CONTEXT       ILH_DLG_MAPPING
#define ILX_HELP_FILEOPEN      ILH_DLG_COMMONOPEN

/*-----
 * Enumerated types.
 *-----*/

typedef enum                    // Field map requests
{

```

```

    ILX_MAP_LAST          = 0,          // Get last saved field map
    ILX_MAP_DEFAULT       = 1,          // Get default field map
    ILX_MAP_VERIFY        = 2,          // Validate field map only
    ILX_MAP_LAST_NOPROMPT = 3           // Get last map but no prompt
} ILX_MAP;

/*-----
 * HIGH-LEVEL function type.
 *-----*/
typedef int (IL_DECL *ILX_PLITEPORT)      // Initiate data transfer
( IL_HWIN,                               // Handle to parent window
  ILX_HAPP,                              // Handle to source system
  ILX_HAPP,                              // Handle to target system
  ILX_PIM );                             // Section types to include

/*-----
 * ILWHATFIELDS and related callback function types.
 *-----*/
typedef int (IL_DECL IL_DIST *PXLFLDS)    // Get field list (ILWhatFields)
( IL_PSTR,                               // Pointer to qualified file name
  ILTB_ID,                               // ID of character map
  ILTB_PHNDL,                            // Pointer to table handle
  IL_HANDLE IL_DIST *,                  // Pointer to field list handle
  int *,                                // Pointer to field count
  char );                               // ASCII delimiter character
typedef int (IL_DECL IL_DIST *PXLFLDSALT) // Extended ILWhatfields call
( ILX_PWFParams );                     // Ptr to WhatFlds Parameter Block
typedef int (IL_DECL IL_DIST *PXLGETPSWD) // Translator ILXGetPassword call
( ILX_PPswdParams );                   // Ptr to params (see ILTR.H)

/*-----
 * LOW-LEVEL function types.
 *-----*/

//----- Add new application section
typedef ILX_HSECTION (IL_DECL *ILX_PADDSECTION)
( ILX_PGLOBALS,                          // Pointer to global data
  ILX_HAPP,                              // Application handle
  IL_PSTR,                               // Pointer to section name
  ILX_PIM,                               // Section type
  ILX_HSECTION * );                     // Pointer to returned handle
typedef int (IL_DECL *ILX_PBEGSESSION)    // Begin translation session
( ILX_PGLOBALS );                       // Pointer to global data
typedef int (IL_DECL *ILX_PENDSESSION)    // End translation session
( ILX_PGLOBALS );                       // Pointer to global data
typedef int (IL_DECL *ILX_PFLIPAPPS)      // Reverse source and target
( ILX_PGLOBALS );                       // Pointer to global data
typedef int (IL_DECL *ILX_PGETERRTEXT)    // Get text of error code
( ILX_PGLOBALS,                          // Pointer to global data
  int,                                    // ILX error code
  IL_PSTR,                               // Pointer to text buffer
  int );                                // Length of text buffer
typedef int (IL_DECL *ILX_PGETFIELDMAP)   // Get field map
( ILX_PGLOBALS,                          // Pointer to global data
  IL_HANDLE IL_DIST *,                   // Pointer to source list handle
  IL_HANDLE IL_DIST *,                   // Pointer to target list handle
  ILPINT,                                // Pointer to source field count
  ILPINT,                                // Pointer to target field count
  int );                                // Request qualifier
typedef int (IL_DECL *ILX_PGETFILENAME)   // Get file name for system
( ILX_PGLOBALS,                          // Pointer to global data
  ILX_HAPP,                              // Application handle
  ILX_HSECTION,                          // Section handle
  int,                                    // File name index
  IL_PSTR,                               // Pointer to returned file name
  int );                                // Length of file name buffer
typedef int (IL_DECL *ILX_PGETSECTIONLIST) // Get list of sections
( ILX_PGLOBALS,                          // Pointer to global data
  ILX_HAPP,                              // Application handle
  ILX_PIM,                               // Section types to include
  IL_HANDLE IL_DIST *,                   // Pointer to list handle
  ILPINT );                             // Pointer to section count
typedef int (IL_DECL *ILX_PGETVALUE)      // Get option value
( ILX_PGLOBALS,                          // Pointer to global data
  ILX_VALUE,                             // Value type

```

```

        long * );
typedef int (IL_DECL *ILX_PMERGE)
    ( ILX_PGLOBALS,
      ILX_OPTION,
      IL_PSTR,
      ILX_BOOL );
typedef int (IL_DECL *ILX_PMERGEX)
    ( ILX_PGLOBALS,
      ILX_ID,
      ILX_OPTION,
      IL_PSTR,
      ILX_BOOL );
typedef int (IL_DECL *ILX_PPUTFIELDMAP)
    ( ILX_PGLOBALS,
      IL_HANDLE,
      IL_HANDLE,
      int,
      int );
typedef int (IL_DECL *ILX_PREADTABLE)
    ( ILX_PGLOBALS,
      ILX_HAPP,
      ILX_TBLOPT,
      long * );
typedef int (IL_DECL *ILX_PREMOVESECTION)
    ( ILX_PGLOBALS,
      ILX_HAPP,
      ILX_HSECTION );
typedef int (IL_DECL *ILX_PRENAMESECTION)
    ( ILX_PGLOBALS,
      ILX_HAPP,
      ILX_HSECTION,
      IL_PSTR );
typedef ILX_PIM (IL_DECL *ILX_PTOPIM)
    ( ILX_PGLOBALS,
      int );
typedef int (IL_DECL *ILX_PSELAPPS)
    ( ILX_PGLOBALS,
      ILX_HAPP,
      ILX_HAPP );
typedef int (IL_DECL *ILX_PSELFIELDS)
    ( ILX_PGLOBALS,
      IL_PSTR,
      IL_PSTR,
      IL_PSTR );
typedef int (IL_DECL *ILX_PSELSECTIONS)
    ( ILX_PGLOBALS,
      ILX_HSECTION,
      ILX_HSECTION );
typedef int (IL_DECL *ILX_PSETFILE)
    ( ILX_PGLOBALS,
      ILX_HAPP,
      ILX_HSECTION,
      int,
      IL_PSTR );
typedef int (IL_DECL *ILX_PSETVALUE)
    ( ILX_PGLOBALS,
      ILX_VALUE,
      long );
typedef int (IL_DECL *ILX_PSHOWFIELDMAP)
    ( ILX_PGLOBALS );
typedef int (IL_DECL *ILX_PSHUTDOWN)
    ( ILX_PGLOBALS );
typedef int (IL_DECL *ILX_PSTARTUP)
    ( ILX_PGLOBALS,
      IL_PSTR,
      IL_HWIN );
typedef int (IL_DECL *ILX_PSTARTUPEX)
    ( ILX_PGLOBALS,
      IL_PSTR,
      IL_HWIN,
      IL_PSTR );
typedef int (IL_DECL *ILX_PTRANSLATE)
    ( ILX_PGLOBALS,
      ILX_OPTION,
      IL_PSTR,
      // Pointer to returned value
      // Perform Merge operation
      // Pointer to global data
      // Reconciliation option
      // Pointer to log file name
      // Show field mapping dialog?
      // Perform Merge operation
      // Pointer to global data
      // Field map ID
      // Reconciliation option
      // Pointer to log file name
      // Show field mapping dialog?
      // Write field map
      // Pointer to global data
      // Handle to source fields
      // Handle to target fields
      // Number of source fields
      // Number of target fields
      // Read data from system table
      // Pointer to global data
      // Application handle
      // Table entry type
      // Pointer to returned entry
      // Remove application section
      // Pointer to global data
      // Application handle
      // Section handle
      // Rename application section
      // Pointer to global data
      // Application handle
      // Section handle
      // New section name
      // Convert section to PIM
      // Pointer to global data
      // Section type
      // Select applications
      // Pointer to global data
      // Source application
      // Target application
      // Select data files
      // Pointer to global data
      // Pointer to source file 1
      // Pointer to source file 2
      // Pointer to target file
      // Select data sections
      // Pointer to global data
      // Source section
      // Target section
      // Set file name
      // Pointer to global data
      // Application handle
      // Section handle
      // File name index
      // Pointer to file name
      // Set option value
      // Pointer to global data
      // Value type
      // Option value
      // Show field mapping dialog
      // Pointer to global data
      // Shut down engine
      // Pointer to global data
      // Start up engine
      // Pointer to global data
      // Pointer to run directory
      // Parent window handle
      // Start up engine
      // Pointer to global data
      // Pointer to run directory
      // Parent window handle
      // Pointer to user name or NULL
      // Perform Translate operation
      // Pointer to global data
      // Reconciliation option
      // Pointer to log file name
    );

```

```

        ILX_BOOL );
typedef int (IL_DECL *ILX_PTRANSLATEX) // Show field mapping dialog?
( ILX_PGLOBALS, // Perform Translate operation
  ILX_ID, // Pointer to global data
  ILX_OPTION, // Field map ID
  IL_PSTR, // Reconciliation option
  ILX_BOOL f; // Pointer to log file name
  // Show field mapping dialog?

//----- Did we find any data errors during translation?
typedef INT16 (IL_DECL *ILX_PTRANSLATEDATAERRORS)
( ILX_PGLOBALS ); // Pointer to global data
typedef int (IL_DECL *ILX_PUPDATETABLE) // Update system record
( ILX_PGLOBALS, // Pointer to global data
  ILX_HAPP, // Application handle
  ILX_TBLOPT, // Table option type
  long ); // Option value
typedef int (IL_DECL *ILX_PFULLBACKUP) // Create a full backup of device
( ILX_PGLOBALS, // Pointer to global data
  ILX_HAPP, // System ID of device to backup
  INT16, // System class of device
  int ); // Com port to use
typedef int (IL_DECL *ILX_PFULLRESTORE) // Restore a full backup to device
( ILX_PGLOBALS, // Pointer to global data
  ILX_HAPP, // System ID of device to restore
  int ); // Com port to use
typedef int (IL_DECL *ILX_PCALLGETPASSWORD) // Call translator ILGetPassword
( ILX_PGLOBALS, // Pointer to ilx globals
  IL_PSTR, // Pointer to translator name
  IL_PSTR, // Pointer to app. file name
  IL_PSTR, // Pointer to app. section name
  IL_PSTR ); // Pointer to password buffer

/*-----
 * EXTENDED function types (added 4/15/95).
 * These functions support creating, deleting, saving, and loading
 * custom field maps.
 *-----*/
typedef ILX_ID (IL_DECL *ILX_PGETACTIVEMAP) // Get active field map
( ILX_PGLOBALS ); // Pointer to global data
typedef int (IL_DECL *ILX_PGETFIELDMAPEX) // Get field map (extended)
( ILX_PGLOBALS, // Pointer to global data
  ILX_ID, // Field map ID to get
  IL_HANDLE IL_DIST *, // Pointer to source list handle
  IL_HANDLE IL_DIST *, // Pointer to target list handle
  ILPINT, // Pointer to source field count
  ILPINT, // Pointer to target field count
  int ); // Request qualifier
typedef int (IL_DECL *ILX_PGETMAPLIST) // Get list of field maps
( ILX_PGLOBALS, // Pointer to global data
  IL_HANDLE IL_DIST *, // Pointer to list handle
  ILPINT ); // Pointer to list item count
typedef int (IL_DECL *ILX_PLOADMAPFILE) // Load field map from file
( ILX_PGLOBALS, // Pointer to global data
  int nType, // Section type to load
  IL_PSTR, // Pointer to file name
  IL_PSTR, // Pointer to field map name
  ILX_ID * ); // Pointer to field map ID
typedef ILX_ID (IL_DECL *ILX_PNEWMAP) // Create new field map
( ILX_PGLOBALS, // Pointer to global data
  ILX_ID, // ID of base field map
  IL_PSTR ); // Pointer to field map name
typedef int (IL_DECL *ILX_PPUTFIELDMAPEX) // Write field map (extended)
( ILX_PGLOBALS, // Pointer to global data
  ILX_ID, // Field map ID to save
  IL_HANDLE, // Handle to source fields
  IL_HANDLE, // Handle to target fields
  int, // Number of source fields
  int ); // Number of target fields
typedef ILX_BOOL (IL_DECL *ILX_PREMOVEMAP) // Remove field map
( ILX_PGLOBALS, // Pointer to global data
  ILX_ID ); // Field map ID to remove
typedef ILX_BOOL (IL_DECL *ILX_PSAVEMAPFILE) // Save field map to file
( ILX_PGLOBALS, // Pointer to global data
  ILX_ID, // Field map ID to save
  IL_PSTR ); // Pointer to file name

```

```

typedef ILX_BOOL (IL_DECL *ILX_PSELECTMAP)    // Set active field map
( ILX_PGLOBS,                                // Pointer to global data
  ILX_ID,                                    // Field map ID to activate
  ILX_BOOL );                               // Set or clear flag?

/*-----
 * Access macros for ILX_GLOBALS structure members.
 *-----*/

#define ILXGL_engineState      (_ilx_globals->engineState)
#define ILXGL_action          (_ilx_globals->action)
#define ILXGL_log              (_ilx_globals->log)
#define ILXGL_bFanRepeat      (_ilx_globals->bFanRepeat)
#define ILXGL_keepFiles       (_ilx_globals->keepFiles)
#define ILXGL_AsciiNames      (_ilx_globals->AsciiNames)
#define ILXGL_remFile         (_ilx_globals->remFile)
#define ILXGL_bFirstXlate     (_ilx_globals->bFirstXlate)
#define ILXGL_nEnviron        (_ilx_globals->nEnviron)
#define ILXGL_apptRange       (_ilx_globals->apptRange)
#define ILXGL_todoRange       (_ilx_globals->todoRange)
#define ILXGL_wrc              (_ilx_globals->wrc)
#define ILXGL_AsciiSep        (_ilx_globals->AsciiSep)
#define ILXGL_nCurDrive      (_ilx_globals->nCurDrive)
#define ILXGL_szCurDir       (_ilx_globals->szCurDir)
#define ILXGL_szDir           (_ilx_globals->szDir)
#define ILXGL_szPswd          (_ilx_globals->szPswd)
#define ILXGL_szSourceFile1    (_ilx_globals->szSourceFile1)
#define ILXGL_szSourceFile2    (_ilx_globals->szSourceFile2)
#define ILXGL_szTargetFile     (_ilx_globals->szTargetFile)
#define ILXGL_hTable          (_ilx_globals->hTable)
#define ILXGL_hSourceApp       (_ilx_globals->hSourceApp)
#define ILXGL_hTargetApp       (_ilx_globals->hTargetApp)
#define ILXGL_hSourceSect      (_ilx_globals->hSourceSect)
#define ILXGL_hTargetSect      (_ilx_globals->hTargetSect)
#define ILXGL_hSourceAppRec     (_ilx_globals->hSourceAppRec)
#define ILXGL_hTargetAppRec     (_ilx_globals->hTargetAppRec)
#define ILXGL_hSourceSectRec    (_ilx_globals->hSourceSectRec)
#define ILXGL_hTargetSectRec    (_ilx_globals->hTargetSectRec)
#define ILXGL_nCommHelpContext  (_ilx_globals->nCommHelpContext)
#define ILXGL_nHelpContext     (_ilx_globals->nHelpContext)
#define ILXGL_nHelpOpen        (_ilx_globals->nHelpOpen)
#define ILXGL_szHelpFile       (_ilx_globals->szHelpFile)
#define ILXGL_bVWRHelp         (_ilx_globals->bVWRHelp)
#define ILXGL_hFileList        (_ilx_globals->hFileList)
#define ILXGL_nFileList        (_ilx_globals->nFileList)
#define ILXGL_bConnected       (_ilx_globals->bConnected)
#define ILXGL_bRamCard         (_ilx_globals->bRamCard)
#define ILXGL_bDocument        (_ilx_globals->bDocument)
#define ILXGL_hCommDLL         (_ilx_globals->hCommDLL)
#define ILXGL_hEngineDLL       (_ilx_globals->hEngineDLL)
#define ILXGL_hSourceDLL       (_ilx_globals->hSourceDLL)
#define ILXGL_hTargetDLL       (_ilx_globals->hTargetDLL)
#define ILXGL_nComPort         (_ilx_globals->nComPort)
#define ILXGL_hWindow          (_ilx_globals->hWindow)
#define ILXGL_hMapWin          (_ilx_globals->hMapWin)
#define ILXGL_hLineWin         (_ilx_globals->hLineWin)
#define ILXGL_hInstance        (_ilx_globals->hInstance)
#define ILXGL_hSessionID       (_ilx_globals->hSessionID)
#define ILXGL_lDateRangeStart  (_ilx_globals->lDateRangeStart)
#define ILXGL_lDateRangeEnd    (_ilx_globals->lDateRangeEnd)
#define ILXGL_nSynchronize     (_ilx_globals->nSynchronize)
#define ILXGL_nXlatorError     (_ilx_globals->nXlatorError)
#define ILXGL_nSystemError     (_ilx_globals->nSystemError)
#define ILXGL_cbProgress       (_ilx_globals->cbProgress)
#define ILXGL_uTimerInterval   (_ilx_globals->uTimerInterval)
#define ILXGL_bCancelRequest    (_ilx_globals->bCancelRequest)
#define ILXGL_nXlateDataErrs   (_ilx_globals->nXlateDataErrs)
#define ILXGL_hProgWin         (_ilx_globals->hProgWin)
#define ILXGL_lTotalRecords    (_ilx_globals->lTotalRecords)
#define ILXGL_bStayConnected   (_ilx_globals->bStayConnected)
#define ILXGL_nConnectedType   (_ilx_globals->nConnectedType)
#define ILXGL_pSourceXtraData   (_ilx_globals->pSourceXtraData)
#define ILXGL_pTargetXtraData   (_ilx_globals->pTargetXtraData)
#define ILXGL_bUseTable        (_ilx_globals->bUseTable)
#define ILXGL_Flags            (_ilx_globals->Flags)
#define ILXGL_hSrcAppSession    (_ilx_globals->hSrcAppSession)

```

```

#define ILXGL_hTarAppSession    (_ilx_globals->hTarAppSession)
#define ILXGL_hOtherSrcDLL      (_ilx_globals->hOtherSrcDLL)
#define ILXGL_hOtherTarDLL      (_ilx_globals->hOtherTarDLL)
#define ILXGL_bSrcWhatFields    (_ilx_globals->bSrcWhatFields)
#define ILXGL_bTarWhatFields    (_ilx_globals->bTarWhatFields)
#define ILXGL_OKTP_Threshold    (_ilx_globals->OKTP_Threshold)
#define ILXGL_hDialogFont      (_ilx_globals->hDialogFont)
#define ILXGL_hExt              (_ilx_globals->hExt)
#define ILXGL_pExt              (_ilx_globals->pExt)
#define ILXGL_version           (_ilx_globals->version)

/*-----
 * Access macros for EXTENDED ILX_GLOBALS structure members.
 *-----*/
#define ILXGL_szUserDir         (_ilx_globals->pExt->szUserDir)

/*-----
 * Macros.
 *-----*/
#define ILX_ENGINE_CHECK() \
    if (ILXGL_engineState == ILX_STATE_DOWN) \
        return ILX_ERR_DOWN;

/*-----
 * General function prototypes.
 *-----*/
void IL_DECL ILX_ClearValues                // Clear certain values from TR
    ( ILTR_PTRANSL tr );                    // Pointer to TR structure
int IL_DECL ILX_DisconnectFromHandheld      // Disconnect from handheld
    ( ILX_PGLOBALS _ilx_globals );         // Pointer to global data
int IL_DECL ILX_EndDriver                   // Terminate translator driver
    ( ILTR_PTRANSL tr,                      // Pointer to translation data
      int nRc );                            // Return code from translator
void IL_DECL ILX_FieldName                  // Format field name for display
    ( ILX_PFIELD pFldName,                 // Pointer to field name
      IL_PSTR pOutName,                    // Pointer to output field name
      int nOutName );                      // Size of output field buffer
void IL_DECL ILX_FreeFieldLists             // Free ILTR field list tables
    ( ILTR_PTRANSL tr );                   // Pointer to translation record
int IL_DECL ILX_GetAppClass                 // Get system class based on ID
    ( ILX_PGLOBALS _ilx_globals,          // Pointer to global data
      ILX_HAPP hApp,                       // Handle for system
      ILTB_CLASS *pnSysClass );            // Pointer to return value
ILX_PIM IL_DECL ILX_GetCapable              // Get system capabilities
    ( ILTB_PSYSREC pRec );                 // Pointer to system record
int IL_DECL ILX_InitDriver                  // Start up translator driver
    ( ILTR_PTRANSL tr );                   // Pointer to translation data
ILX_BOOL IL_DECL ILX_IsCapable              // Is section type supported?
    ( ILX_PIM nCapable,                    // System capabilities
      ILTB_SEC nType );                    // Section type
void IL_DECL ILX_KillStatusMeter            // Destroy status dialog
    ( ILX_PGLOBALS _ilx_globals );         // Pointer to global data
int IL_DECL ILX_LoadCommDLL                 // Load communication DLL
    ( ILX_PGLOBALS _ilx_globals,          // Pointer to global data
      ILTB_TYPE SysType,                  // System type
      ILTB_CLASS SysClass,                // System class
      int nComPort,                       // Communication port
      BOOL16 bConnect );                  // Initiate connection as well?
int IL_DECL ILX_LoadFieldLists              // Load field list/mapping info
    ( ILTR_PTRANSL tr,                     // Ptr to translation structure
      ILTB_PHNDL phTable );               // Ptr to ITB open file handle
int IL_DECL ILX_LoadTranslator              // Call xlator to import or export
    ( ILX_PGLOBALS _ilx_globals,          // Pointer to global data
      ILTR_PTRANSL tr,                     // Pointer to translation record
      IL_PSTR pExeFile );                 // Appl. executable file name
int IL_DECL ILX_MapILTRErrorCode            // Convert ILTR to ILX error code
    ( int nError );                       // ILTR error code to convert
int IL_DECL ILX_PutLogHeader                // Write header info to log
    ( ILX_PGLOBALS _ilx_globals,          // Pointer to global data
      ILX_ACTION nAction,                 // Import or export action
      ILX_OPTION nReconcile,              // Reconciliation option
      IL_HFILE IL_DIST *hFile );          // Pointer to map file handle
int IL_DECL ILX_RemoveCommDLL              // Remove communication DLL
    ( ILX_PGLOBALS _ilx_globals );         // Pointer to global data
int IL_DECL ILX_RetrieveFieldList           // Retrieve field list

```



```

    ( ILTB_PHNDL phFile,
      ILTB_ID nListID,
      IL_HANDLE *phFields,
      ILPINT pnFields );
int IL_DECL ILX_RetrieveFieldMap
    ( ILX_PGLOBALS _ilx_globals,
      ILTB_PHNDL phTable,
      ILTB_ID nMapID,
      ILTB_ID *pnSrcList,
      ILTB_ID *pnTarList,
      IL_HANDLE *hMap,
      ILTB_PMAP *pMap );
void IL_DECL ILX_ResetCurrentDir
    ( ILX_PGLOBALS _ilx_globals );
void IL_DECL ILX_SetCurrentDir
    ( ILX_PGLOBALS _ilx_globals );
int IL_DECL ILX_ShowStatusMeter
    ( ILTR_PTRANSL tr,
      ILX_PGLOBALS _ilx_globals );
void IL_DECL ILX_SortField
    ( INT16 IL_DIST *nTop,
      int ndx,
      ILX_PFIELD pField,
      ILTR_SORT_ORDER nOrder );
int IL_DECL ILX_StartApp
    ( IL_PSTR pExeFile,
      IL_PSTR pAppName,
      int nExeFile,
      IL_HWIN hParentWnd,
      IL_HINST hParentInst,
      UINT *hModule );
int IL_DECL ILX_TellTIFAboutFields
    ( ILTR_PTRANSL tr );
int ILX_ValidateSource
    ( ILTB_PSYSREC pRec,
      IL_PSTR pFileName );
int ILX_ValidateTarget
    ( ILTB_PSYSREC pRec,
      IL_PSTR pFileName );
int ILX_VerifyFileName
    ( ILTB_PSYSREC pRec,
      IL_PSTR pFileName );
ILX_BOOL IL_DECL ILX_VerifyFieldMap
    ( ILX_PGLOBALS _ilx_globals,
      ILX_BOOL *bMapOK );

// Pointer to table handle
// Field list ID
// Pointer to field list handle
// Pointer to number of fields
// Get field map from table
// Pointer to global data
// Pointer to table handle
// Map ID
// Pointer to source list ID
// Pointer to target list ID
// Pointer to field map handle
// Pointer to field map
// Reset working directory
// Pointer to global data
// Set working directory
// Pointer to global data
// Show status dialog
// Pointer to translation data
// Pointer to global data
// Perform logical sort on field
// Pointer to top index
// Index of current field
// Pointer to field record
// Logical sort order
// Load Windows application
// Pointer to executable file name
// Pointer to system name
// Size of executable file name
// Window handle
// Instance handle
// Module handle
// Load field names into TIF
// Pointer to translation record
// Validate source file name
// Pointer to base system record
// Pointer to file name
// Validate target file name
// Pointer to base system record
// Pointer to file name
// Validate DOS file name
// Pointer to base system record
// Pointer to file name
// Validate field lists and map
// Pointer to global data
// Pointer to valid flag

/*-----
 * Routines for loading/unloading/calling a translator DLL or Code resource.
 *-----*/
int IL_DECL ILX_LoadXlator
    ( IL_PSTR pDirName,
      IL_PSTR pXlatorName,
      IL_PXULATOR phXlator );
// Load translator (.FIL) module
// Ptr to engine directory name
// Xlator name from system table
// Ptr to translator "handle"
void IL_DECL ILX_UnloadXlator
    ( IL_PXULATOR phXlator );
// Unload/release external xlator
// Ptr to translator "handle"
int IL_DECL ILX_CallWhatFields
    ( IL_PSTR pszAppFile,
      IL_LPHANDLE phFldList,
      ILPINT pnFldCount,
      ILX_PGLOBALS _ilx_globals,
      ILTB_PSYSREC pSystemRec,
      IL_PANY pXtraData );
// Setup and call ILWhatFields
// Ptr to data file name
// Ptr to prior field list handle
// Ptr to prior field count
// Ptr to ilx globals
// Ptr to system record
// Ptr to translators xtra data
FARPROC IL_DECL ILX_GetXlateProc
    ( IL_PXULATOR phXlator,
      IL_PSTR pProcName );
// Get ptr to translator function
// Ptr to translator "handle"
// Ptr to function name

/*-----
 * Windows callback routines.
 *-----*/
#ifdef ILWIN
BOOL IL_DECL EXP ILX_DlgFieldMap
    ( HWND hDlg,
      UINT nMessage,
      WPARAM wParam,
      LPARAM lParam );
// Show field mapping dialog
// Window handle
// Windows message ID
// First parameter

```



```
        LPARAM lParam );
BOOL IL_DECL_EXP ILX_DlgProgress
(   HWND hDlg,
    UINT nMessage,
    WPARAM wParam,
    LPARAM lParam );

// Second parameter
// Show progress dialog
// Window handle
// Windows message ID
// First parameter
// Second parameter

#endif // ILWIN

//----- Special handling for C++ code
#ifdef __cplusplus
}
#endif // __cplusplus

#endif // __ILXAPI
```

```

/*-----
* Name:      LOADFLDS.C
*
* Purpose:   Functions to load and initialize Field Lists, Field Mapping,
*            Character Mapping info and SST (Section SubType) lists,
*            from TABLES.ITB into the tr structure.
*
* Functions: ILX_LoadFieldLists -- Initial load of field info from TABLES.ITB
*            ILX_FreeFieldLists -- Final deallocation of field information
*            ILX_TellTIFAboutFields -- Load source/target fields names to TIF
*
* Internal:  GetFieldLists ----- Loads and initializes field list info.
*            FieldListSanityCheck -- does a sanity check
*            LoadCharMap ----- Loads a specific Character Mapping table
*            GetSSTList ----- Gets List of Section SubTypes
*            TweakSST ----- Decides SST Behavior for current engine run
*            HasOneTaggedField -- Find TAGGED Field in field list
*            PseudoMapTaggedField - Set MapField==ILX_UNMAPPED_BUT_TAGGED
*
* Author:    Bob Daley, Copyright (c) IntelliLink, 1995
*
* Notes:     ILX_LoadFieldLists is called ONCE from XLATE.C or XLATEW.C for
*            a full Import, Export, or SyncPort cycle. ILX_LoadFieldLists is
*            called BEFORE any calls are made to any specific translator.
*            ILExport and ILImport make use of the table information left in
*            the tr structure by ILX_LoadFieldLists, so that we don't need to
*            re-initialize this information from TABLES.ITB for each import
*            or Export operation involved in the cycle. ILX_FreeFieldLists
*            is called once at the conclusion of the cycle to clean up.
*
*            ILX_LoadFieldLists may also be called from ILX16. When this is
*            the case, the source and target system may be reversed, as ILX16
*            may be operating in ILTR_PHASE10 or ILTR_PHASE40.
*-----*/

```

```
#include "ilxapi.h"
```

```
//----- Internal functions
```

```

static
int IL_DECL GetFieldLists          // Load field list/mapping information
( ILTR_PTRANSL tr,                // Pointer to translation structure
  ILTB_PHNDL phTable,             // Pointer to ITB open file handle
  ILTB_ID nSourceSystem,          // Original source system ID
  ILTB_ID nTargetSystem,          // Original target system ID
  ILTB_ID nSourceSection,         // Original source section ID
  ILTB_ID nTargetSection );       // Original target section ID

static
int IL_DECL FieldListSanityCheck
( ILTR_FLDPTR List,
  INT16 Count );

static
int IL_DECL GetSSTList            // Get list of Section SubTypes
( ILTR_PTRANSL tr,                // Pointer to translation structure
  ILTB_PHNDL phTable,             // Pointer to ITB open file handle
  ILTB_ID sysid,                  // ID of system
  ILTB_ID secid,                  // ID of current section
  BYTE IL_DIST *pCurrentSST,      // where to squirrel away current SST
  ILTR_PSSTLIST pList );         // Ptr to List structure to be filled in

static
int IL_DECL LoadCharMap           // Load specified CharMap table
( ILTB_PHNDL phTable,             // Pointer to ITB open file handle
  ILTB_ID nCharMapID,             // ITB character map table ID
  ILTR_PBUFFER pCharBuff );       // Pointer to CharMap buffer

static
int TweakSST (ILTR_PTRANSL tr, BYTE SourceSST, BYTE TargetSST);

static
int HasOneTaggedField             // Find TAGGED field in field list
( ILTR_PTRANSL tr,                // Pointer to translation structure
  ILTR_FLDPTR List,               // Ptr to field list
  ILTR_NDX TopIndex );            // Starting point for scan of field list

static
int PseudoMapTaggedField          // Set MapField==ILX_UNMAPPED_BUT_TAGGED
( ILTR_FLDPTR List,               // Ptr to field list

```

```

        ILTR_NDX TopIndex );          // Starting point for scan of field list

/*-----
 * Name:      ILX_LoadFieldLists
 *
 * Purpose:   Load and initialize all field list, field mapping, character
 *            mapping, and SST (Section SubType) lists into the tr structure
 *            prior to translation.
 *
 * Called by: ilx_v3/xlate.c/doTranslate
 *            ilwin/xlatew.c/DoTranslate
 *            ilx16/ilx16.cpp/ReloadTableInfo
 *
 * Input:     tr ----- Pointer to translation structure
 *            phTable -- Pointer to open TABLES.ITB file.
 *
 * Return:    SUCCESS or ILTR error code
 *-----*/

int IL_DECL ILX_LoadFieldLists      , // Load field list/mapping information
    ( ILTR_PTRANSL tr,              // Pointer to translation structure
      ILTB_PHNDL phTable )          // Pointer to ITB open file handle
{
    int          nRc;                // Fuction return code
    BOOLEAN      bRepeat = FALSE;    // TRUE if repeat fields need to be added
    ILTR_FIELD   sExtraField;        // Template for "extra" (hidden) fields
    ILTR_FLDPTR  pS, pT;             // Pointers to source/target field entries
    ILTR_PFLDMAP pFields;            // Pointer to ILTR field map structure
    ILTB_ID      nSourceSystem;       // Original source system
    ILTB_ID      nTargetSystem;       // Original target system
    ILTB_ID      nSourceSection;      // Original source section
    ILTB_ID      nTargetSection;      // Original target section
    BYTE         SourceSST;           // Original source section subtype
    BYTE         TargetSST;           // Original target section subtype

    //----- Complain if running under a stone age app or engine (xlate.c) build
    if (ILTR_VERSION_IS_PRIOR_TO (14))
        return ILERROR(ILTR_version, ILTR_ERR_STONE_AGE_APP);

    /*-----
     * If the current translation phase is Phase10 or Phase40 then the source
     * and target systems have already been reversed.  This can happen when we
     * are called from ILX16.  When the source and target are reversed, we need
     * to determine the ORIGINAL source and target of this translation in order
     * to load the field lists properly.
     *-----*/
    if (ILTR_phase == ILTR_PHASE10 || ILTR_phase == ILTR_PHASE40)
    {
        //----- Source and Target are reversed. Get the ORIGINAL source and target
        nSourceSystem = ILTR_nTargetID;
        nTargetSystem = ILTR_nSourceID;
        nSourceSection = ILTR_nTarSection;
        nTargetSection = ILTR_nSrcSection;
    }
    else
    {
        //----- Source and Target not reversion. Use source and target from tr.
        nSourceSystem = ILTR_nSourceID;
        nTargetSystem = ILTR_nTargetID;
        nSourceSection = ILTR_nSrcSection;
        nTargetSection = ILTR_nTarSection;
    }

    //----- Allocate space for the Field Map and Field List table information
    IL_ALLOC_MEM (sizeof (ILTR_TABLEINFO), ILTR_hTableInfo, ILTR_pTableInfo);
    if (ILTR_pTableInfo == NULL)
        return ILERROR (sizeof(ILTR_TABLEINFO), ILTR_ERR_NOMEM);

    IL_MEMSET (ILTR_pTableInfo, 0, sizeof (ILTR_TABLEINFO));

    //----- Save the ORIGININAL source/target system ID's.
    ILTR_pTableInfo->nOriginalSourceID = nSourceSystem;
    ILTR_pTableInfo->nOriginalTargetID = nTargetSystem;

```

```

//----- Increase count to accommodate _appData and _subType fields
ILTR_pTableInfo->nExtraFields += ILTR_EXTRA_FIELDS_ALWAYS;

//----- Add extra fields to field count for repeat information fields
if ( ILTR_nFunction == ILTR_APPT ||
      ILTR_nFunction == ILTR_TODO ||
      ILTR_nFunction == ILTR_CALL )
{
    ILTR_pTableInfo->nExtraFields += ILTR_EXTRA_FIELDS_FOR_REPEAT;
    bRepeat = TRUE;
}

//----- Load and initialize field lists structures from TABLES.ITB file
nRc = GetFieldLists ( tr, phTable, nSourceSystem, nTargetSystem,
                     nSourceSection, nTargetSection );

if (nRc)
    return nRc;

//----- Build List of Source SSTs (Section SubTypes)
nRc = GetSSTList ( tr, phTable,
                  nSourceSystem,
                  nSourceSection,
                  &SourceSST,
                  &ILTR_pTableInfo->sOriginalSourceSSTList );

if (nRc)
    return nRc;

//----- Build List of Target SSTs (Section SubTypes)
nRc = GetSSTList ( tr, phTable,
                  nTargetSystem,
                  nTargetSection,
                  &TargetSST,
                  &ILTR_pTableInfo->sOriginalTargetSSTList );

if (nRc)
    return nRc;

/*-----
 * If SST isn't already disabled, decide how SST shall behave for the
 * current engine run. The decision is based on ILTR_Flags and
 * Source & Target Section SubTypes and existence and mapped/unmapped
 * status of Source & Target TAGGED Fields. Unmapped Tagged fields may
 * be tweaked to appear pseudo-mapped if pseudo-mapping is not prohibited.
 *-----*/
if ((ILTR_Flags & ILTR_DISABLE_SST_TAGGING) == 0)
{
    nRc = TweakSST (tr, SourceSST, TargetSST);
    if (nRc)
        return nRc;
}

//----- Establish pointer to ILTR_FLDMAP structure
pFields = &ILTR_pTableInfo->sFieldMap;

//----- Establish field list "template" for all "extra" (hidden) fields
sExtraField.ItemNo      = 1;
sExtraField.Type        = (char) ILX_TYPE_BINARY;
sExtraField.Width       = 0;
sExtraField.Term        = ILX_TERM_EOS;
sExtraField.Attrbts     = ILTB_ATT_HIDDEN_FIELD | ILTB_ATT_NO_RECONCILE;
/*-----
 * All of the standard hidden fields (appData & Repeat basic+excl)
 * are implicitly "mapped" in the sense that the data in these fields
 * is carried across from Source to Target. But in field info structure
 * we mark these fields with 'ILX_UNMAPPED' because we don't need a
 * field number to keep track of the implicit mapping.
 *-----*/
sExtraField.MapField     = ILX_UNMAPPED;
sExtraField.Assoc       = 0;
sExtraField.Index       = -1;
sExtraField.NextField   = -1;
sExtraField.PriorField  = -1;
sExtraField.Label[0]    = '\0';
sExtraField.TypeDesc[0] = '\0';
sExtraField.IntName[0]  = '\0';
IL_STRCPY (sExtraField.ExtName, "~");

```

```

//----- Get pointer to first extra field list item (after last "real" item)
pS = &pFields->pSource[pFields->nSource];
pT = &pFields->pTarget[pFields->nTarget];

//----- Add the extra field for application data ("_appData") to source list
IL_STRCPY (sExtraField.IntName, ILTR_APP_DATA);
*pS++ = sExtraField;
*pT++ = sExtraField;

//----- Add repeat fields only if we've established these fields are needed
if (bRepeat)
{
    //----- Add the "basic" repeat field ("_repBasic") to source list
    IL_STRCPY (sExtraField.IntName, ILTR_REP_BASIC);
    *pS++ = sExtraField;
    *pT++ = sExtraField;

    //----- Add the "Exclusion List" repeat field ("_repExcl") to source list
    IL_STRCPY (sExtraField.IntName, ILTR_REP_XDATE);
    *pS++ = sExtraField;
    *pT++ = sExtraField;
}

//----- Add the extra field for SST ("_subType") to source list
IL_STRCPY (sExtraField.IntName, ILTR_SUB_TYPE);
sExtraField.Type = (char) ILX_TYPE_NUMBER;
sExtraField.Attrb = ILTB_ATT_HIDDEN_FIELD | ILTB_ATT_KEY_FIELD;
*pS++ = sExtraField;
*pT++ = sExtraField;

//----- Load the character mapping tables
nRc = LoadCharMap ( phTable, ILTR_nSourceExportCharMapID,
                    &ILTR_pTableInfo->sSourceExportCharMap );
if (nRc)
    return ILTR_ERR_NOCHARMAP;

nRc = LoadCharMap ( phTable, ILTR_nSourceImportCharMapID,
                    &ILTR_pTableInfo->sSourceImportCharMap );
if (nRc)
    return ILTR_ERR_NOCHARMAP;

nRc = LoadCharMap ( phTable, ILTR_nTargetExportCharMapID,
                    &ILTR_pTableInfo->sTargetExportCharMap );
if (nRc)
    return ILTR_ERR_NOCHARMAP;

nRc = LoadCharMap ( phTable, ILTR_nTargetImportCharMapID,
                    &ILTR_pTableInfo->sTargetImportCharMap );
if (nRc)
    return ILTR_ERR_NOCHARMAP;

//----- All done. Return success
return SUCCESS;
}

/*-----
* Name:      ILX_FreeFieldLists
* Purpose:   Free the field list and CharMap information allocated by
*            ILX_LoadFieldLists
* Input:     tr ----- Pointer to translation structure
* Return:    void
*-----*/

void IL_DECL ILX_FreeFieldLists // Free ILTR field list/mapping information
( ILTR_PTRANSL tr )           // Pointer to translation structure
{
    //----- Ignore the call if there is nothing to be freed
    if (ILTR_VERSION_IS_PRIOR_TO (14) || ILTR_pTableInfo == NULL)
        return;

    //----- Free the source and target field lists
    if (ILTR_pTableInfo->sFieldMap.pSource)
        IL_FREE_AND_ZERO ( ILTR_pTableInfo->sFieldMap.hSource,

```

```

        ILTR_pTableInfo->sFieldMap.pSource );
if (ILTR_pTableInfo->sFieldMap.pTarget)
    IL_FREE_AND_ZERO ( ILTR_pTableInfo->sFieldMap.hTarget,
        ILTR_pTableInfo->sFieldMap.pTarget );

//----- Free the Export and Import Character Maps
if (ILTR_pTableInfo->sSourceExportCharMap.buffer)
    IL_FREE_AND_ZERO ( ILTR_pTableInfo->sSourceExportCharMap.handle,
        ILTR_pTableInfo->sSourceExportCharMap.buffer );
if (ILTR_pTableInfo->sSourceImportCharMap.buffer)
    IL_FREE_AND_ZERO ( ILTR_pTableInfo->sSourceImportCharMap.handle,
        ILTR_pTableInfo->sSourceImportCharMap.buffer );
if (ILTR_pTableInfo->sTargetExportCharMap.buffer)
    IL_FREE_AND_ZERO ( ILTR_pTableInfo->sTargetExportCharMap.handle,
        ILTR_pTableInfo->sTargetExportCharMap.buffer );
if (ILTR_pTableInfo->sTargetImportCharMap.buffer)
    IL_FREE_AND_ZERO ( ILTR_pTableInfo->sTargetImportCharMap.handle,
        ILTR_pTableInfo->sTargetImportCharMap.buffer );

//----- Finally, free the table information structure
IL_FREE_AND_ZERO (ILTR_hTableInfo, ILTR_pTableInfo);
}

/*-----
* Name:      ILX_TellTIFAboutFields
* Purpose:   Load either the source or target field names into TIF.
* Input:     tr ----- Pointer to translation structure
*            phTable -- Pointer to open TABLES.ITB file.
* Return:    SUCCESS or ILTR error code
*-----*/

int IL_DECL ILX_TellTIFAboutFields // Load source/target field names into TIF
    ( ILTR_PTRANSL tr )           // Pointer to translation structure
{
#ifdef ILTIF_IN_ENGINE

    ILTR_NDX      i;                // Loop/index variable
    int           nRc;              // Fuction return code
    ILTR_NDX      nList;            // Number of source/target fields in list
    ILTR_FLDPTR   pList;            // Pointer to source or target field list

    //----- Set up field lists for current ILTR_nSource/ILTR_nTarget settings
    nRc = ILSetupTables (tr);
    if (nRc)
        return nRc;

    //----- Get pointer and field count for export or import translator
    if (ILTR_phase == ILTR_PHASE05)
    {
        //----- Use field list of exporting translator
        nList = ILTR_pTableInfo->sFieldMap.nSource;
        pList = ILTR_pTableInfo->sFieldMap.pSource;
    }
    else
    {
        //----- Use field list of importing translator
        nList = ILTR_pTableInfo->sFieldMap.nTarget;
        pList = ILTR_pTableInfo->sFieldMap.pTarget;
    }

    //----- Add the number of special extra fields to the field count
    nList += ILTR_pTableInfo->nExtraFields;

    //----- Tell TIF about all "top level" fields
    for (i = 0; i < nList; ++i)
    {
        //----- Skip any sub-fields
        if (pList[i].ItemNo != 1)
            continue;

        //----- Provide TIF field descriptor
        nRc = ILTIFDefFieldN (tr, i, 0);
        if (nRc != SUCCESS)
            return nRc;
    }
}

```

```

    }

    /*-----
    * When finished telling TIF about Target Fields (which is always
    * done after telling TIF about Source Fields), tell TIF that we're
    * entirely done telling it about fields. This causes TIF to
    * analyze all the field info, which may well lead to detection
    * of a fatal error condition...
    *-----*/
    if (ILTR_phase == ILTR_PHASE10)
    {
        nRc = ILTIFStartNextPhase(tr, TIF_PHASE_DONE_SETTING_THINGS_UP);
        if (nRc != SUCCESS)
            return nRc;
    }

    //----- Fields successfully defined to TIF. Free the field list tables
    ILFreeTables (tr);

#endif // ILTIF_IN_ENGINE

    //----- All done.
    return SUCCESS;
}

/*-----
* Name:      GetFieldLists
*
* Purpose:   Load and initialize field list and field mapping information
*            into the tr structure prior to translation.
*
* Caller:    ILX_LoadFieldLists
*
* Input:     tr ----- Pointer to translation structure
*            phTable ----- Pointer to open TABLES.ITB file.
*            nSourceSystem --- Original source system ID
*            nTargetSystem --- Original target system ID
*            nSourceSection -- Original source section ID
*            nTargetSection -- Original target section ID
*
* Return:    SUCCESS or ILTR error code
*
* Note:      Code adapted from ILFldLoadMap by Mike Blanchette (LOADMAP.C).
*-----*/
static
int IL_DECL GetFieldLists          // Load field list/mapping information
(
    ILTR_PTRANSL tr,              // Pointer to translation structure
    ILTB_PHNDL phTable,          // Pointer to ITB open file handle
    ILTB_ID nSourceSystem,        // Original source system ID
    ILTB_ID nTargetSystem,        // Original target system ID
    ILTB_ID nSourceSection,       // Original source section ID
    ILTB_ID nTargetSection        // Original target section ID
)
{
    ILTR_NDX i;                  // Loop variable
    int error = SUCCESS;         // Return code
    int fieldCount;              // Field count
    int len;                     // Record length
    int memSize;                 // Memory size
    int ndx;                     // Field index
    int rc = SUCCESS;            // Return code
    ILTB_ID sourceFldID;          // field list ID of source list in file
    ILTB_ID targetFldID;          // field list ID of target list in file
    IL_HANDLE hMap;              // Handle to field map
    IL_HANDLE hSource;           // Handle to source field list
    IL_HANDLE hTarget;           // Handle to target field list
    ILTB_PMAP pMap = NULL;        // Pointer to map record
    ILTR_PFLDMAP pFields;         // Pointer to ILTR field map structure
    ILTB_PFLDLST pSource = NULL;  // Pointer to source field list
    ILTB_PFLDLST pTarget = NULL;  // Pointer to target field list

    //----- Establish pointer to ILTR Table info copy of ILTR_FLDMAP structure
    pFields = &ILTR_pTableInfo->sFieldMap;

    //----- Initialize variables.

```



```

pFields->nSource      = pFields->nTarget      = 0;
pFields->ShowSource   = pFields->ShowTarget   = -1;
pFields->TopSource    = pFields->TopTarget    = -1;
pFields->ApptDate     = -1;
pFields->pSource       = pFields->pTarget     = NULL;
pFields->nMapStatus   = 0;
pFields->sName[0]     = '\0';

/*-----
 * Find a suitable field map ID in ITB tables file if no Map ID is
 * specified in the translation structure.
 *-----*/
if (!ILTR_nMapID)
{
    if (!(ILTR_nMapID = ILTBFindMapID ( phTable,
                                       nSourceSystem, nSourceSection,
                                       nTargetSystem, nTargetSection,
                                       &ILTR_reversed )))
    {
        error = ILERROR (0, ILTR_ERR_NOMAP);
        goto errorExit;
    }
}

//----- Now obtain the map record length.
len = ILTBGetMapRecLen (phTable, ILTR_nMapID);
if (! len)
{
    error = ILERROR (0, ILTR_ERR_BADMAP);
    goto errorExit;
}

//----- Allocate sufficient memory to load map record in memory.
IL_ALLOC_MEM (len, hMap, pMap);
if (pMap == NULL)
{
    error = ILERROR (len, ILTR_ERR_NOMEM);
    goto errorExit;
}

//----- Read the map record.
if (ILTBGetMapRec (phTable, ILTR_nMapID, NULL, pMap, len))
{
    error = ILERROR (0, ILTR_ERR_BADMAP);
    goto errorExit;
}

//----- Check if systems are reversed in this field map.
if (ILTBGetFldAppID (phTable, pMap->sourceList) == nSourceSystem)
    ILTR_reversed = FALSE;
else
    ILTR_reversed = TRUE;

//----- Note the map name and number of mapped fields.
IL_STRCPY (pFields->sName, pMap->name);
fieldCount = ILTBGetFldFieldNum (phTable, pMap->sourceList);

//----- Set the source and target field list ID's based on order.
if (ILTR_reversed)
{
    sourceFldID = pMap->targetList;
    targetFldID = pMap->sourceList;
}
else
{
    sourceFldID = pMap->sourceList;
    targetFldID = pMap->targetList;
}

//----- Determine the size of the field list.
len = ILTBGetFldRecLen (phTable, sourceFldID);
if (! len)
{
    error = ILERROR (0, ILTR_ERR_BADMAP);
    goto errorExit;
}

```

```

}

//----- Allocate memory required to hold all source fields.
IL_ALLOC_MEM (len, hSource, pSource);
if (pSource == NULL)
{
    error = ILERROR (len, ILTR_ERR_NOMEM);
    goto errorExit;
}

//----- Now read the field list into memory.
if (ILTBGetFldRec (phTable, sourceFldID, pSource, len))
{
    error = ILERROR (0, ILTR_ERR_BADMAP);
    goto errorExit;
}

//----- Post the field associations in the source fields.
for (i = 0; i < fieldCount; i++)
{
    if ((ndx = pMap->mapIndex[i]) != ILX_UNMAPPED)
    {
        if (ILTR_reversed)
            pSource->field[ndx].mapIndex = i;
        else pSource->field[i].mapIndex = ndx;
    }
}

/*-----
 * Now allocate memory for the source list in a format that will actually
 * be used by the translators. Also allocate room for "extra" fields.
 *-----*/
pFields->nSource = pSource->fieldNum;
memSize = sizeof (ILTR_FIELD) *
            (pFields->nSource + ILTR_pTableInfo->nExtraFields);
IL_ALLOC_MEM (memSize, pFields->hSource, pFields->pSource);
if (pFields->pSource == NULL)
{
    error = ILERROR (memSize, ILTR_ERR_NOMEM);
    goto errorExit;
}
IL_MEMSET (pFields->pSource, 0, memSize);

//----- Process all source fields.
for (i = 0; i < pFields->nSource; i++)
{
    //----- Copy all fields from input record to output buffer.
    pFields->pSource[i].ItemNo      = pSource->field[i].itemNo;
    pFields->pSource[i].Type       = (char) pSource->field[i].type;
    pFields->pSource[i].Width      = pSource->field[i].width;
    pFields->pSource[i].Term       = pSource->field[i].delimit;
    pFields->pSource[i].Attribs    = pSource->field[i].attribs;
    pFields->pSource[i].MapField   = pSource->field[i].mapIndex;
    pFields->pSource[i].Assoc      = pSource->field[i].assoc;
    pFields->pSource[i].Index      = -1;
    pFields->pSource[i].NextField  = -1;
    pFields->pSource[i].PriorField = -1;
    IL_STRCPY (pFields->pSource[i].IntName, pSource->field[i].label);
    IL_STRCPY (pFields->pSource[i].ExtName, pSource->field[i].name);
    IL_STRCPY (pFields->pSource[i].Label, pSource->field[i].prefix);
    IL_STRCPY (pFields->pSource[i].TypeDesc, pSource->field[i].typeDesc);

    //----- Locate insertion point and post next/prior field indexes.
    ILFldInsert (&pFields->TopSource, i, pFields->pSource);
}

//----- Do a quick sanity-check of the source field list
rc = FieldListSanityCheck (pFields->pSource, pFields->nSource);
if (rc != SUCCESS)
{
    error = ILERROR (rc, ILTR_ERR_BADMAP);
    goto errorExit;
}

//----- Free the source list record.

```

```

if (pSource)
{
    IL_FREE_MEM (hSource, pSource);
    pSource = NULL;
}

//----- Determine the size of the target field list.
len = ILTBGetFldRecLen (phTable, targetFldID);
if (! len)
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

//----- Allocate memory required to hold all target field descriptors.
IL_ALLOC_MEM (len, hTarget, pTarget);
if (pTarget == NULL)
{
    error = ILTR_ERR_NOMEM;
    goto errorExit;
}

//----- Now read target field list into memory.
if (ILTBGetFldRec (phTable, targetFldID, pTarget, len))
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

//----- Post the field associations in the target fields.
for (i = 0; i < fieldCount; i++)
{
    if ((ndx = pMap->mapIndex[i]) != ILX_UNMAPPED)
    {
        if (ILTR_reversed)
            pTarget->field[i].mapIndex = ndx;
        else pTarget->field[ndx].mapIndex = i;
    }
}

/*-----
 * Now allocate memory for the target list in a format that will actually
 * be used by the translators. Also allocate room for "extra" fields.
 *-----*/
pFields->nTarget = pTarget->fieldNum;
memSize = sizeof (ILTR_FIELD) *
            (pFields->nTarget + ILTR_pTableInfo->nExtraFields);
IL_ALLOC_MEM (memSize, pFields->hTarget, pFields->pTarget);
if (pFields->pTarget == NULL)
{
    error = ILTR_ERR_NOMEM;
    goto errorExit;
}
IL_MEMSET (pFields->pTarget, 0, memSize);

//----- Process all source fields.
for (i = 0; i < pFields->nTarget; i++)
{
    //----- Copy all fields from input record to output buffer.
    pFields->pTarget[i].ItemNo      = pTarget->field[i].itemNo;
    pFields->pTarget[i].Type       = (char) pTarget->field[i].type;
    pFields->pTarget[i].Width      = pTarget->field[i].width;
    pFields->pTarget[i].Term       = pTarget->field[i].delimit;
    pFields->pTarget[i].Attribs    = pTarget->field[i].attribs;
    pFields->pTarget[i].MapField   = pTarget->field[i].mapIndex;
    pFields->pTarget[i].Assoc      = pTarget->field[i].assoc;
    pFields->pTarget[i].Index      = -1;
    pFields->pTarget[i].NextField  = -1;
    pFields->pTarget[i].PriorField = -1;
    IL_STRCPY (pFields->pTarget[i].IntName, pTarget->field[i].label);
    IL_STRCPY (pFields->pTarget[i].ExtName, pTarget->field[i].name);
    IL_STRCPY (pFields->pTarget[i].Label, pTarget->field[i].prefix);
    IL_STRCPY (pFields->pTarget[i].TypeDesc, pTarget->field[i].typeDesc);

    //----- Locate insertion point and post next/prior field indexes.

```

```

    ILFldInsert (&pFields->TopTarget, i, pFields->pTarget);
}

//----- Do a quick sanity-check of the target field list
rc = FieldListSanityCheck (pFields->pTarget, pFields->nTarget);
if (rc != SUCCESS)
{
    error = ILERROR (rc, ILTR_ERR_BADMAP);
    goto errorExit;
}

errorExit:

//----- Free dynamic memory before leaving.
if (pMap)
    IL_FREE_MEM (hMap, pMap);
if (pSource)
    IL_FREE_MEM (hSource, pSource);
if (pTarget)
    IL_FREE_MEM (hTarget, pTarget);

//----- Return with error code (default == SUCCESS).
return (error ? error : rc);

} //----- GetFieldLists

/*-----
 * Name:      FieldListSanityCheck -- called from GetFieldLists
 *-----*/
static
int IL_DECL FieldListSanityCheck (ILTR_FLDPTR List, INT16 Count)
{
    INT16 PrevItemNo = -99;
    INT16 ThisItemNo;
    INT16 i;

    for (i = 0; i < Count; i++)
    {
        ThisItemNo = List[i].ItemNo;
        if (List[i].Attribs & ILTB_ATT_COMBINED)
        {
            //----- All combined fields must have ItemNo==1
            if (ThisItemNo == 1)
                PrevItemNo = 1;
            else
                return ILERROR ((int) i, ILTR_ERR_BADMAP);
        }

        //----- A regular (NOT combined) field with ItemNo==1 is OK anywhere,
        //----- but let's make sure it isn't followed by a field with ItemNo <> 1
        else if (ThisItemNo == 1)
            PrevItemNo = -99;

        //----- If ItemNo is NOT 1, demand a smooth upward succession of Item Numms
        else if (ThisItemNo == PrevItemNo + 1)
            PrevItemNo = ThisItemNo;

        //----- Else complain about out-of-sequence itemNo...
        else
            return ILERROR ((int) i, ILTR_ERR_BADMAP);
    }

    return SUCCESS;
} //----- FieldListSanityCheck

/*-----
 * Name:      LoadCharMap
 *
 * Purpose:   Load requested character mapping table if one is specified and
 *            provided. Otherwise, load the default character mapping table.
 *
 * Caller:    ILX_LoadFieldLists
 *-----*/

```

```

*
* Input:      phTable -- Pointer to ITB open file handle
*             CharMap -- Character map ID
*             pBuffer -- Pointer to ILTR_BUFFER structure to hold CharMap
* Return:     SUCCESS, FAILURE, or ILTB error code
* Author:     Bob Daley, Copyright (c) IntelliLink, 1995
*-----*/
static
int IL_DECL LoadCharMap          // Load specified CharMap table
( ILTB_PHNDL phTable,           // Pointer to ITB open file handle
  ILTB_ID nCharMapID,           // ITB character map table ID
  ILTR_PBUFFER pCharBuff )     // Pointer to CharMap buffer
{
    int i;                      // Temporary loop/index variable
    int iRc;                    // Function return code
    int nMemSize;               // Number of bytes to be allocated
    IL_HANDLE hCharMap;         // Handle for CharMap record buffer
    ILTB_PCHARMAPREC pCharMap = NULL; // Pointer to CharMap record buffer

    //----- Allocate memory for character mapping table.
    nMemSize = ILTB_CHARMAP_CHARS * sizeof (char);
    IL_ALLOC_MEM (nMemSize, pCharBuff->handle, pCharBuff->buffer);
    if (pCharBuff->buffer == NULL)
        return ILTR_ERR_NOMEM;

    //----- Initialize null CharMap table in case no CharMap provided.
    for (i = 0; i < ILTB_CHARMAP_CHARS; i++)
        pCharBuff->buffer[i] = i;

    //----- If no CharMap ID provided, simply return success.
    if (nCharMapID == 0)
        return SUCCESS;

    //----- Make sure we have a valid CharMap ID.
    if (! ILTBIsCharMapRec (phTable, nCharMapID))
    {
        //----- Simply ignore error if this ITB file doesn't support CharMaps.
        iRc = ILTBLastError (phTable);
        if (iRc == ILTB_ERR_VERSION)
            return SUCCESS;

        //----- Not an "ITB file version" issue. We have a bad CharMap ID.
        return ILTR_ERR_NOCHARMAP;
    }

    //----- Get the record length for this CharMap ID.
    nMemSize = ILTBGetCharMapRecLen (phTable, nCharMapID);
    if (nMemSize == 0)
        return ILTR_ERR_MAPFILE;

    //----- Allocate memory for the FULL CharMap record.
    IL_ALLOC_MEM (nMemSize, hCharMap, pCharMap);
    if (pCharMap == NULL)
        return ILTR_ERR_NOMEM;

    //----- Read in the complete CharMap record.
    iRc = ILTBGetCharMapRec (phTable, nCharMapID, pCharMap, nMemSize);
    if (iRc)
    {
        iRc = ILTR_ERR_NOCHARMAP;
        goto CleanUpAndReturn;
    }

    //----- Initialize the Character Mapping table from the CharMap record data.
    for (i = 0; i < ILTB_CHARMAP_CHARS; i++)
        pCharBuff->buffer[i] = pCharMap->cChars[i];

    //----- All done, indicate successful completion.
    iRc = SUCCESS;

    //----- Clean up and return status.
CleanUpAndReturn:
    if (pCharMap)
        IL_FREE_MEM (hCharMap, pCharMap);
    return iRc;
}

```

```

}

/*-----
* Name:      GetSSTList
*
* Purpose:   Build list of all SSTs (Section SubType), for the specified
*            system, for the current Section Type (ILTR_nFunction).
*
* Caller:    ILX_LoadFieldLists
*
* Return:    SUCCESS or ILTR error code
*
* Author:    David Boothby, Copyright (c) IntelliLink, 1996
*-----*/
static
int IL_DECL GetSSTList ( ILTR_PTRANSL tr,
                        ILTB_PHNDL phTable,
                        ILTB_ID sysid,
                        ILTB_ID secid,
                        BYTE IL_DIST *pCurrentSST,
                        ILTR_PSSTLIST pList )
{
    ILTB_SEC SectionType = ILTR_nFunction;
    int rc;
    int nLen;
    int i;
    IL_HANDLE hSysRec = IL_NULL_HANDLE;    // Handle to system record
    ILTB_PSYSREC pSysRec = NULL;           // Pointer to system record

    //----- initialize count of subtypes
    pList->sstCount = 0;

    //----- Get size of complete system record
    nLen = ILTBGetSysRecLen (phTable, sysid);
    if (!nLen)
        return ILERROR (nLen, ILTR_ERR_ILTB_ERROR);

    //----- Allocate buffer to read system record into
    IL_ALLOC_MEM (nLen, hSysRec, pSysRec);
    if (pSysRec == NULL)
        return ILTR_ERR_NOMEM;

    //----- Read complete system record
    rc = ILTBGetSysRec (phTable, sysid, pSysRec, nLen);
    if (rc != ILTB_OK)
    {
        IL_FREE_MEM (hSysRec, pSysRec);
        return ILERROR(rc, ILTR_ERR_ILTB_ERROR);
    }

    //----- Find the desired section in system record
    for (i = 0; i < pSysRec->SectionCount; i++)
    {
        //----- Is this one of the desired sections?
        if (pSysRec->Section[i].SectionType == SectionType)
        {
            //----- Make sure section is active
            if (!(pSysRec->Section[i].SectionAttrib & ILTB_ATT_ACTIVE))
                continue;

            //----- Copy SubType of current section
            if (pSysRec->Section[i].SectionID == secid)
                *pCurrentSST = (BYTE) pSysRec->Section[i].SectionSubType;

            //----- skip if list is already FULL. Could complain here!!
            if (pList->sstCount >= ILTR_MAX_SST_COUNT)
                continue;

            //----- Add this section's subtype to the SST List
            pList->sstList[pList->sstCount++] =
                (BYTE) pSysRec->Section[i].SectionSubType;
        }
    }
}

```

```

    IL_FREE_MEM (hSysRec, pSysRec);
    return SUCCESS;
}

/*-----
* Name:      TweakSST
*
* Purpose:   Decide how SST shall behave for the current engine run
*
* Caller:    ILX_LoadFieldLists
*
* Method:    Make sure that all pre-requisites are met for applying the
*            SST mechanism.  If not, turn it off.  If any tagged fields
*            are unmapped, and if pseudo-mapping isn't prohibited, then
*            tweak the in-memory field map to make such fields be
*            pseudo-mapped.
*
*            Pseudo-mapped fields have MapField==ILX_UNMAPPED_BUT_TAGGED.
*            Almost all IntelliLink modules are duped into believing that
*            pseudo-mapped fields are truly MAPPED.  But TIF and
*            the ILTR/fldget.c/ILFldGetEx function are wiser.
*
* Return:    SUCCESS or ILTR error code
*
* Author:    David Boothby, Copyright (c) IntelliLink, 1996
*-----*/
static
int TweakSST (ILTR_PTRANSL tr, BYTE SourceSST, BYTE TargetSST)
{
    ILTR_PFLDMAP pFields = &ILTR_pTableInfo->sFieldMap;
    int sHas = FALSE;
    int tHas = FALSE;
    int nRc;

    //---- Verify that at least one Section SubType (Source or Target) is MAIN.
    if ( (SourceSST != ILX_SUBSECT_MAIN)
        && (TargetSST != ILX_SUBSECT_MAIN) )
    {
        //---- Neither Source nor Target is MAIN; turn off SST
        ILTR_Flags |= (ILTR_DISABLE_SST_TAGGING | ILTR_DISABLE_SST_FILTERING);
        return SUCCESS;
    }

    //---- Make sure Source Field List has a Tagged Field if it needs one
    if (SourceSST == ILX_SUBSECT_MAIN)
    {
        //---- Check for Tagged Field; make sure it's mapped if required
        sHas = HasOneTaggedField (tr, pFields->pSource, pFields->TopSource);
        if (sHas == FALSE)
        {
            //--- Source lacks required [mapped] tagged field; turn off SST
            ILTR_Flags |= (ILTR_DISABLE_SST_TAGGING | ILTR_DISABLE_SST_FILTERING);
            return SUCCESS;
        }
        else if (sHas != TRUE && sHas != ILTR_ERR_NOTMAPPED)
            return sHas;          // abnormal error
    }

    //---- Make sure Target Field List has a Tagged Field if it needs one
    if (TargetSST == ILX_SUBSECT_MAIN)
    {
        //---- Check for Tagged Field; make sure it's mapped if required
        tHas = HasOneTaggedField (tr, pFields->pTarget, pFields->TopTarget);
        if (tHas == FALSE)
        {
            //--- Target lacks required [mapped] tagged field; turn off SST
            ILTR_Flags |= (ILTR_DISABLE_SST_TAGGING | ILTR_DISABLE_SST_FILTERING);
            return SUCCESS;
        }
        else if (tHas != TRUE && tHas != ILTR_ERR_NOTMAPPED)
            return tHas;          // abnormal error
    }

    //---- If Source needs pseudo-mapping, do it now

```



```

    if (sHas == ILTR_ERR_NOTMAPPED)
    {
        nRc = PseudoMapTaggedField (pFields->pSource, pFields->TopSource);
        if (nRc != SUCCESS)
            return nRc;
    }

    //---- If Target needs pseudo-mapping, do it now
    if (tHas == ILTR_ERR_NOTMAPPED)
    {
        nRc = PseudoMapTaggedField (pFields->pTarget, pFields->TopTarget);
        if (nRc != SUCCESS)
            return nRc;
    }

    //---- All set. SST mechanism is enabled and ready to run.
    return SUCCESS;
}

/*-----
* Name:      HasOneTaggedField
*
* Purpose:   Scan field list looking for TAGGED fields. There should be
*            either zero or one of them. Complain if there are more than 1.
*            Also complain if a sub-item is TAGGED. The TAGGED attribute
*            should only be applied to top-level fields.
*            Check whether tagged field is mapped or not and decide outcome
*            based on that and on whether pseudo-mapping is allowed or not.
*
* Caller:    TweakSST
*
* Return:    TRUE if one mapped tagged field is found
*            FALSE if no tagged fields are found or if tagged field
*            is unmapped and pseudo-mapping is prohibited
*            ILTR_ERR_NOTMAPPED if pseudo-mapping is allowed and
*            one unmapped tagged field is found
*            ILTR_ERR_TOO_MANY_TAGGED if more than one tagged field is found
*            ILTR_ERR_SUB_ITEM_TAGGED if a tagged sub-item is found.
*
* Author:    David Boothby, Copyright (c) IntelliLink, 1996
*-----*/
static
int HasOneTaggedField (ILTR_PTRANSL tr, ILTR_FLDPTR List, ILTR_NDX TopIndex)
{
    ILTR_NDX cur = TopIndex;           // Index to current field
    ILTR_NDX taggedField = -1;         // Index to tagged field, when found

    //---- Find ALL tagged fields in list (there shouldn't be more than one)
    while (cur != -1)
    {
        if (List[cur].Attribs & ILTB_ATT_TAGGED)
        {
            if (taggedField != -1)
                //---- Found more than one tagged field. That's bad.
                return ILTR_ERR_TOO_MANY_TAGGED;

            if (List[cur].ItemNo == 1)
                //---- Found FIRST tagged field. Remember it.
                taggedField = cur;
            else
                //---- Found a tagged field that isn't a top-level field!
                return ILTR_ERR_SUB_ITEM_TAGGED;
        }

        cur = List[cur].NextField;
    }

    if (taggedField == -1)
        //---- Didn't find any tagged fields.
        return FALSE;

    if (List[taggedField].MapField != ILX_UNMAPPED)
        //---- Top-level tagged field is mapped.
        return TRUE;
}

```

```

//---- Check to see if any sub-items of tagged field are mapped
cur = List[taggedField].NextField;
while (cur != -1)
{
    if (List[cur].ItemNo == 1)
    /*-----
     * ItemNo==1 means that we've stepped beyond the last sub-item of
     * the tagged field; look no further for mapped sub-items.
     *-----*/
        break;

    if (List[cur].MapField != ILX_UNMAPPED)
        //---- Tagged field has a mapped sub-item. That's good enough!
        return TRUE;

    cur = List[cur].NextField;
}

//---- Tagged field is UNMAPPED. Is that OK?
if (ILTR_Flags & ILTR_DISABLE_SST_IF_UNMAPPED)
    //---- Pseudo-mapping prohibited, so an unmapped tagged field has same
    //---- affect on SST operation as would absolute lack of a tagged field
    return FALSE;
else
    //---- Pseudo-mapping allowed; indicate that it is needed!
    return ILTR_ERR_NOTMAPPED;
}

/*-----
 * Name:      PseudoMapTaggedField
 *
 * Purpose:   Find and pseudo-map the first TAGGED field in a given field list
 *            by changing its MapField setting to ILX_UNMAPPED_BUT_TAGGED.
 *
 * Caller:    TweakSST
 *
 * NOTE:      Caller is responsible for verifying that the first TAGGED field
 *            in the field list is UNMAPPED before calling this function.
 *
 *            It isn't sensible to have more than one TAGGED field in a
 *            Field List, but we don't check for that aberration here.
 *
 * Return:    SUCCESS or ILTR error code
 *
 * Author:    David Boothby, Copyright (c) IntelliLink, 1996
 *-----*/
static
int PseudoMapTaggedField (ILTR_FLDPTR List, ILTR_NDX TopIndex)
{
    ILTR_NDX cur = TopIndex;          // Index to current field in List

    //---- scan Field List until we succeed or hit end of list
    while (cur != -1)
    {
        if (List[cur].Attribs & ILTB_ATT_TAGGED)
        {
            //---- found TAGGED field; psuedo-map it.
            List[cur].MapField = ILX_UNMAPPED_BUT_TAGGED;
            return SUCCESS;
        }

        cur = List[cur].NextField;
    }

    //---- bad news: hit end of list
    return ILERROR (-1, ILTR_ERR_INTERNAL_ERROR);
}

```

```

/*-----
* Name:      XLATE.C
* Purpose:   Perform complete translation operation
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1994
*
* Entrypoints:  ILX_dTranslate and ILX_dTranslateEx
*
* Local Functions:
*
*      GetReady
*      LoadCommIfNeeded
*      CheckOrShowFieldMap
*      Set_ILTR_Members
*      Set_trFlags
*      StartLogFile
*      InitProgressDisplay
*      PerformTranslations
*      SetPhaseParams
*      MapTIFReturnCode
*      ILX_ILTIFInit
*      ILX_ILTIFClose
*      ILX_ILTIFSyncInit
*      ILX_ILTIFSyncFinishUpAndClose
*      ILX_ILTIFStartNextPhase
*      TellTIFAboutFields
*      ILX_ILTIFCloseFileInitially
*      ILX_ILTIFReopenFile
*      doTranslate
*      Choose_V3_V4_Or_Sync
*      SyncFromScratchIfFilesAreNew
*      do_Phase05_Setup
*      do_Phase10_ExportFromTarget
*      LabelProgressBars
*      do_Phase20_ExportFromSource
*      do_Phase30_ImportIntoTarget
*      do_Phase40_ImportIntoSource
*      do_SyncFinishUp
*      GetRemoteInputFile
*      PreFetchTargetFile
*      FindTempFile
*      RemoveTargetFileJustOnce
*      FlHookProc
*      KillFlHookProc
*      MakeFlHookProc
*
* Notes:
*   This file contains the core routines necessary to perform
*   a full translation operation. The translator modules must
*   have been loaded before the call to ILX_dTranslate.
*-----*/
#include "ilxapi.h"                // Main API header

//----- Global variables
#ifdef ILWIN

//----- Windows only variables
unsigned long _nHelpContext;      // Help context number
char _szHelpFile[ILTB_MAX_PATH]; // Help file name
ILX_BOOL _bVWRHelp;              // Use Viewer Help?
HHOOK _hHook=0;                  // Handle to Hook

//----- Windows only "hook"
LRESULT CALLBACK FlHookProc      // Message hook
( int nCode,                    // Input event
  WPARAM wParam,                // Unused
  LPARAM lParam );              // Address of MSG structure
static void KillFlHookProc      // Remove Fl Hook
( void );                      // Unused
static int MakeFlHookProc       // Create Fl Hook
( HINSTANCE hInst );           // Instance handle

#endif // ILWIN

//----- Handheld temp files

```

```

static char szTempRemoteSource[MAX_PATH];
static char szTempRemoteTarget[MAX_PATH];

//----- Local symbolic constants
#define MAX_FILE_PREFIX    3

//----- Macros
#define CHECK_FOR_SHARP_WIZARD(nClass, n)
{
    if (nClass == ILTB_CLASS_WIZARD)
        n = ILX_ERR_COM_SHARP;
}
#define EXIT_WITH_ERROR(n) { nRc = n; goto Exit; }

//----- Function prototypes for local routines
static int GetReady
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  ILX_OPTION nOption );

#ifdef ILWIN
static int LoadCommIfNeeded
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr );
#endif

static int CheckOrShowFieldMap
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILX_ID nMapId,
  ILX_BOOL bShowMap );

static int Set_ILTR_Members
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr );

static UINT32 Set_trFlags
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz );

static int StartLogFile
( ILX_PGLOBALS _ilx_globals,
  ILTR_PTRANSL tr,
  IL_PSTR pLogName );

#ifdef ILWIN
static int InitProgressDisplay
( ILX_PGLOBALS _ilx_globals,
  ILTR_PTRANSL tr );
#endif

static int PerformTranslations
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  UINT32 trFlags,
  int EnvAttribs );

static int SetPhaseParams
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvironmentAttribs,
  char phase );
// Set ILTR params for current phase
// Global data
// ptr to set of ptrs to system records
// Translation structure
// Environmental System Attributes
// one of the ILTR_PHASEnn #defines

static int doTranslate
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  UINT32 trFlags,
  );
// Perform Import and Export pair
// Global data
// ptr to set of ptrs to system records
// Translation structure
// initial value for ILTR_Flags

```

```

    int EnvAttribs );                // Environmental System Attributes

static int Choose_V3_V4_Or_Sync
( ILX_PRECS prz,
  ILTR_PTRANSL tr );

static void SyncFromScratchIfFilesAreNew
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr );

static int do_Phase05_Setup
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs );

static int do_Phase10_ExportFromTarget
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs );

#ifdef ILWIN
#ifndef WIN32
static void LabelProgressBars
( ILX_PGLOBALS _ilx_globals,
  ILTR_PTRANSL tr,
  IL_PSTR szSourceFile,
  IL_PSTR szTargetFile,
  ILTB_ACC nSourceAccessType,
  ILTB_ACC nTargetAccessType );
#endif
#endif

static int do_Phase20_ExportFromSource
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs );

static int do_Phase30_ImportIntoTarget
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs );

static int do_Phase40_ImportIntoSource
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs );

static int do_SyncFinishUp
( ILTR_PTRANSL tr );

static int PreFetchTargetFile
( ILX_PGLOBALS _ilx_globals,          // Global data
  ILTR_PTRANSL tr,                   // Translation structure
  ILTB_PSYSREC pTarSys );            // Pointer to Target sysrec

static int GetRemoteInputFile
( ILX_PGLOBALS _ilx_globals,          // Global data
  ILTR_PTRANSL tr,                   // Translation structure
  ILTB_PSYSREC pSys,                 // Pointer to Target sysrec
  IL_PSTR szRemoteFileName,          // Remote File Name
  int nPutWhere );                  // Is this source or target?

static int FindTempFile
( ILX_PGLOBALS _ilx_globals,          // Global data
  IL_PSTR lpOriginal,                // Original file name
  IL_PSTR lpTemp,                    // Returned file name
  ILX_BOOL *bExist,                 // Does file already exist?
  INT16 nPutWhere );                // one of the ILX_PUT_XXX #defines

```

```

static int RemoveTargetFileJustOnce
( ILX_PGLOBALS _ilx_globals,      // Global data
  ILTR_PTRANSL tr );              // Translation structure

/*-----
* Name:      ILX_dTranslate
* Purpose:   Perform complete translation operation
* Parameters:
*   _ilx_globals - Pointer to global data
*   nOption - Reconciliation option
*   pLogFile - Pointer to log file name
*   bMapFields - Show Field Mapping dialog box?
* Returns:   ILX_OK or error code
*-----*/
int IL_DECL ILX_dTranslate ( ILX_PGLOBALS _ilx_globals,
                             ILX_OPTION nOption,
                             IL_PSTR pLogName,
                             ILX_BOOL bShowMap )
{
    //----- Call extended function without a field map ID
    return (ILX_dTranslateEx (_ilx_globals, 0, nOption, pLogName, bShowMap));
}

/*-----
* Name:      ILX_dTranslateEx
* Purpose:   Perform complete translation operation
* Parameters:
*   _ilx_globals - Pointer to global data
*   nMapID - Map ID or zero for default map
*   nOption - Reconciliation option
*   pLogFile - Pointer to log file name
*   bMapFields - Show Field Mapping dialog box?
* Returns:   ILX_OK or error code
*-----*/
int IL_DECL ILX_dTranslateEx ( ILX_PGLOBALS _ilx_globals,
                              ILX_ID nMapId,
                              ILX_OPTION nOption,
                              IL_PSTR pLogName,
                              ILX_BOOL bShowMap )
{
    int nRc;
    char szLoc[ILTB_MAX_PATH];           // Application location
    ILTR_TRANSL trs;                     // Translation information
    ILTR_PTRANSL tr = &trs;              // pointer to trs
    ILX_RECS rz;                         // set of ptrs to system records
    ILX_PRECS prz = &rz;                 // ptr to set of ptrs to system records
    UINT32 trFlags;                      // translation flags for doTranslate
    int EnvAttribs;                      // Environmental System Attributes

    //---- do sanity checking and lay the groundwork
    nRc = GetReady (_ilx_globals, prz, tr, nOption);
    if (nRc != SUCCESS)
        goto Exit;

    #ifdef ILWIN
        //---- load the ILCOMM DLL if necessary
        nRc = LoadCommIfNeeded (_ilx_globals, prz, tr);
        if (nRc != SUCCESS)
            goto Exit;
    #endif

    //---- Check or Show Field Map
    nRc = CheckOrShowFieldMap (_ilx_globals, prz, nMapId, bShowMap);
    if (nRc != SUCCESS)
        goto Exit;

    /*-----
    * Figure out what Environment Attribute bits, if any, we'll
    * need to OR into the ILTR_nAttribs at each stage of translation.
    *-----*/
    switch (ILXGL_nEnviron)
    {
        case ILX_ENV_WINPAD:              EnvAttribs = ILTB_ATT_WINPAD; break;
    }
}

```

```

    case ILX_ENV_ACHATES:
    case ILX_ENV_LITE:
    case ILX_ENV_MAGICXCHANGE: EnvAttribs = ILTB_ATT_LITE;    break;

    case ILX_ENV_NORMAL:      EnvAttribs = 0;                break;

    default: //---- unknown environment type
              nRc = ILERROR (ILXGL_nEnviron, ILX_ERR_INTERNAL);
              goto Exit;
}

//---- Set "tr" attribs (w/o source/target system attribs for now)
ILTR_nAttribs = EnvAttribs;

//---- Set fieldmap ID in "tr" structure
ILTR_nMapID = nMapId;

//---- Copy various parameters into the "tr" structure
nRc = Set_ILTR_Members (_ilx_globals, prz, tr);
if (nRc != SUCCESS)
    goto Exit;

//---- Set the "trFlags" variable (which eventually becomes ILTR_Flags)
trFlags = Set_trFlags (_ilx_globals, prz);

//---- Create or append to logfile, if logging is enabled
nRc = StartLogFile (_ilx_globals, tr, pLogName);
if (nRc != SUCCESS)
    goto Exit;

//----- Initialize driver for translation
nRc = ILX_InitDriver (tr);
if (nRc != SUCCESS)
    goto Exit;

#ifdef ILWIN
    //---- Set up status meter that will be used to show progress
    nRc = InitProgressDisplay (_ilx_globals, tr);
    if (nRc != SUCCESS)
        goto Exit;
#endif

//---- Make one or more calls to "doTranslate"
nRc = PerformTranslations (_ilx_globals, prz, tr, trFlags, EnvAttribs);

/*-----
 * Check to see if we need to update the application location fields
 * for either source or target system. If the user was prompted to
 * change the application location, we want to reflect the latest
 * location in the system record. If the location was changed,
 * function ILX_LoadTranslator will have placed the changes in the
 * SrcLoc and TarLoc members of the ILX_RECS structure.
 *-----*/
if (nRc == ILX_OK)
{
    //----- Is there a source application location?
    if (IL_STRLen (prz->pSrcApp->AppLoc))
    {
        //----- Get the current source location value
        nRc = ILX_dReadTable ( _ilx_globals,
                               prz->pSrcApp->SysID,
                               ILX_TBL_APPLOC,
                               (long *) szLoc );

        //----- Update it only if changed
        if (nRc == ILX_OK && IL_STRCOMP (szLoc, prz->pSrcApp->AppLoc))
        {
            nRc = ILX_dUpdateTable ( _ilx_globals,
                                     prz->pSrcApp->SysID,
                                     ILX_TBL_APPLOC,
                                     (long) prz->pSrcApp->AppLoc );
        }
    }
}

```



```

//----- Is there a target application location?
if (IL_STRLEN (prz->pTarApp->AppLoc))
{
    //----- Get the current target location value
    nRc = ILX_dReadTable ( _ilx_globals,
                          prz->pTarApp->SysID,
                          ILX_TBL_APPLOC,
                          (long *) szLoc );

    //----- Update it only if changed
    if (nRc == ILX_OK && IL_STRCOMP (szLoc, prz->pTarApp->AppLoc))
    {
        nRc = ILX_dUpdateTable ( _ilx_globals,
                                prz->pTarApp->SysID,
                                ILX_TBL_APPLOC,
                                (long) prz->pTarApp->AppLoc );
    }
}

Exit:

ILX_EndDriver (tr, ILTR_rc);

#ifdef ILWIN
    KillF1HookProc ();
#endif

if ( (nRc != SUCCESS)
    && (nRc != ILX_ERR_CANCEL)
    && (nRc != ILX_ERR_NODATA) )
{
    /*-----
    * Record error details in ILERRORS.LOG.
    * Please do not remove or #ifdef this code.
    *-----*/
    ILERROR ( _ilx_globals->nSystemError,
              (int) (tr == NULL ? -1 : ILTR_phase) );
    ILERROR ( _ilx_globals->nXlatorError, nRc);
}

//----- Reset first translate flag so we create the progress bar only once
ILXGL_bFirstXlate = FALSE;

//----- Save and return status
ILXGL_wrc = nRc;
return nRc;
} //---- ILX_dTranslateEx

/*-----
* GetReady (called from ILX_dTranslateEx)
*-----*/
static int GetReady
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  ILX_OPTION nOption )
{
    int nRc = SUCCESS;
    ILX_ACCESS nAccess;           // Access type
    int tmp;
    ILX_BOOL bResetOption = FALSE; // Reset reconcile option?

    #ifdef ILWIN
        _hHook=0;                // Have no handle to F1 Hook yet
    #endif

    /*-----
    * Force global error code to indicate error until we successfully
    * complete translation. This error code is used in END to determine
    * whether we should proceed with remote data transfer to the handheld.
    * If any errors have occurred here, then data transfer will not occur.
    *-----*/

```

```

ILXGL_wrc = ILX_NOTOK;

//----- Clear the translation record
IL_MEMSET (tr, '\0', sizeof (ILTR_TRANSL));

//----- Set default operation to TRANSLATE unless called from MERGE
if (ILXGL_action != ILX_ACT_MERGE)
    ILXGL_action = ILX_ACT_TRANSLATE;

//----- Make sure that session has been activated
if (!(ILXGL_engineState & ILX_STATE_ACTIVE))
    return ILX_ERR_INACTIVE;

//----- Make sure that source and target applications have been set
if (!ILXGL_hSourceApp || !ILXGL_hTargetApp)
    return ILX_ERR_APP;

//----- Make sure that source and target sections have been set
if (!ILXGL_hSourceSect || !ILXGL_hTargetSect)
    return ILX_ERR_SECTION;

//----- Get pointer to system and section records
prz->pSrcApp = (ILTB_PSYSREC) GlobalLock (ILXGL_hSourceAppRec);
prz->pSrcSec = (ILTB_PSECREC) GlobalLock (ILXGL_hSourceSectRec);
prz->pTarApp = (ILTB_PSYSREC) GlobalLock (ILXGL_hTargetAppRec);
prz->pTarSec = (ILTB_PSECREC) GlobalLock (ILXGL_hTargetSectRec);

/*-----
 * Set the file name to the section code if this is a device
 * that does not support files.
 *-----*/
nAccess = prz->pSrcSec->AccessType;
if (nAccess == ILX_ACCESS_NONE)
    IL_STRCPY (ILXGL_szSourceFile1, prz->pSrcSec->Code);

//----- Clear file name if using DDE to communication with application
else if (nAccess == ILX_ACCESS_DDE)
    ILXGL_szSourceFile1[0] = '\0';

/*-----
 * Make sure that source file has been set when the access type
 * is not set to DDE or NONE.
 *-----*/
if (nAccess != ILX_ACCESS_DDE && nAccess != ILX_ACCESS_NONE)
    if (ILXGL_szSourceFile1[0] == '\0')
        return ILX_ERR_FILES;

/*-----
 * Set the file name to the section code if this is a device
 * that does not support files.
 *-----*/
nAccess = prz->pTarSec->AccessType;
if (nAccess == ILX_ACCESS_NONE)
    IL_STRCPY (ILXGL_szTargetFile, prz->pTarSec->Code);

//----- Clear the file name is using DDE to access application data
else if (nAccess == ILX_ACCESS_DDE)
    ILXGL_szTargetFile[0] = '\0';

/*-----
 * Make sure that target file has been set when the access type
 * is not set to DDE.
 *-----*/
if (nAccess != ILX_ACCESS_DDE && nAccess != ILX_ACCESS_NONE)
    if (ILXGL_szTargetFile[0] == '\0')
        return ILX_ERR_FILES;

if (ILXGL_nSynchronize == ILXTR_SYNC_NO)
{
    //--- as a DEMO-staging convenience, allow a back door way to
    //--- do synchronization
    tmp = GetProfileInt("ilwin", "synchronize", ILXTR_SYNC_NO);
    if (tmp < 100)
        ILXGL_nSynchronize = (ILXTR_SYNC_OPTION) tmp;
}

```

```

//--- as a DEMO-staging convenience, allow INI file to override the
//--- conflict resolution option specified by the calling app.
tmp = GetProfileInt("ilwin", "ILCROption", 100);
if (tmp < 100)
    nOption = (ILX_OPTION) tmp;

//----- Validate the option setting
if (ILXGL_nSynchronize == ILXTR_SYNC_NO)
{
    if (nOption != ILX_OPT_REPLACE    &&
        nOption != ILX_OPT_IGNORE    &&
        nOption != ILX_OPT_NOTIFY    &&
        nOption != ILX_OPT_INSERT    &&
        nOption != ILX_OPT_ACCEPT_1  &&
        nOption != ILX_OPT_ACCEPT_2  &&
        nOption != ILX_OPT_UPDATE    &&
        nOption != ILX_OPT_DELETE    && // weird hack for Intel
        nOption != ILX_OPT_NONE)
        return ILX_ERR_OPT;
}
else
{
    /*-----
    * Fix up CR option if it isn't specified correctly for Synchro-
    * nization. You can get ADD_ACROSS behavior either by specifying
    * ILX_OPT_INSERT (preferred) or ILX_OPT_UPDATE (tolerated).
    *-----*/
    if (nOption == ILX_OPT_UPDATE)
        nOption = ILX_OPT_INSERT;           // TIFSCRO_ADD_ACROSS

    if ( (nOption != ILX_OPT_IGNORE)       // TIFSCRO_LEAVE_UNRESOLVED
        && (nOption != ILX_OPT_NOTIFY)      // TIFSCRO_NOTIFY
        && (nOption != ILX_OPT_INSERT)      // TIFSCRO_ADD_ACROSS
        && (nOption != ILX_OPT_ACCEPT_1)    // TIFSCRO_TARGET_WINS
        && (nOption != ILX_OPT_ACCEPT_2) )  // TIFSCRO_SOURCE_WINS
        return ILX_ERR_OPT;
}

/*-----
* Override the supplied reconciliation option if the caller has
* specified that system table options should be used and we are
* NOT performing a synchronization operation. For synchronization
* we ALWAYS use the caller's reconciliation option which is NOT
* equivalent to the normal import/export conflict settings.
*-----*/
if (ILXGL_bUseTable && ILXGL_nSynchronize == ILXTR_SYNC_NO)
    nOption = prz->pTarApp->ReconcileOpt;
else
{
    /*-----
    * Here we validate that the requested reconciliation option is
    * supported by the selected target system. This is to avoid
    * the problem of the caller always specifying the reconciliation
    * option to NOTIFY, for example, without recognizing that not
    * all translators support this option. If the specified option
    * is not available with the selected target system, then we
    * revert the reconcile option that the default value in the
    * system record.
    *-----*/
    switch (nOption)
    {
        case ILX_OPT_REPLACE:
            break;
        case ILX_OPT_IGNORE:
            break;
        case ILX_OPT_NOTIFY:
            if (!(prz->pTarApp->SysAttrib & ILTB_ATT_NOTIFY))
                bResetOption = TRUE;
            break;
        case ILX_OPT_INSERT:
            if (!(prz->pTarApp->SysAttrib & ILTB_ATT_INSERT))
                bResetOption = TRUE;
            break;
        case ILX_OPT_UPDATE:
    }
}

```

```

        if (!(prz->pTarApp->SysAttrib & ILTB_ATT_UPDATE))
            bResetOption = TRUE;
        break;
    case ILX_OPT_MERGE:
        if (!(prz->pTarApp->SysAttrib & ILTB_ATT_MERGE))
            bResetOption = TRUE;
        break;
    case ILX_OPT_ACCEPT_1:
    case ILX_OPT_ACCEPT_2:
        break;
    case ILX_OPT_NONE:
        if (!(prz->pTarApp->SysAttrib & ILTB_ATT_SHOWNONE))
            bResetOption = TRUE;
        break;
    case ILX_OPT_DELETE:
        break;
}

//----- Do we need to reset the reconcile option?
if (bResetOption == TRUE)
{
    if (ILXGL_nSynchronize == ILXTR_SYNC_NO)
        nOption = prz->pTarApp->ReconcileOpt;
    else
        return ILX_ERR_OPT;
}
}

/*-----
 * Finally, if running in "silent mode", change Notify to Add. Silent mode
 * is intended for use by "remote" users who cannot respond to any dialogs.
 *-----*/
if (ILXGL_Flags & ILX_FLAG_UNATTENDED_MODE && nOption == ILX_OPT_NOTIFY)
    nOption = ILX_OPT_INSERT;

//----- Set the update option in the "tr" structure
ILTR_nUpdOpt = nOption;

//----- Turn off the option to stay connected if both systems are handhelds.
if ( prz->pSrcApp->SysType != ILTB_TYPE_APP &&
    prz->pTarApp->SysType != ILTB_TYPE_APP )
    ILXGL_bStayConnected = ILX_FALSE;

//----- If building for Windows, prepare for WinHelp
#ifdef ILWIN

    //----- Set Help context and file name
    _nHelpContext = ILXGL_nHelpOpen;
    _bVWRHelp = ILXGL_bVWRHelp;
    IL_STRCAT (IL_STRCPY (_szHelpFile, ILXGL_szDir), IL_FILESEP_STR);
    IL_STRCAT (_szHelpFile, ILXGL_szHelpFile);

    //----- Create Hook function to trap F1 key presses
    nRc = MakeF1HookProc (hDLLInstance);

#endif //ILWIN

return nRc;
} //----- GetReady

//----- Skip over Comm. support if building for the Mac
#ifdef ILWIN

/*-----
 * LoadCommIfNeeded (called from ILX_dTranslateEx)
 *-----*/
static int LoadCommIfNeeded
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr )
{
    int nRc = SUCCESS;
    ILX_BOOL bLoadComm = FALSE;      // Load communication module?

```

```

int nComPort = 0;                // System communication port
ILTB_TYPE nSysType = 0;          // System type
ILTB_TYPE nSysClass = 0;         // System class

/*-----
 * Determine if we need to load communication module now. We always
 * load the communication DLL if the source system is a handheld device.
 * We also load it if the target system is a handheld which supports
 * data reconciliation (requiring pre-fetching of data from handheld).
 *-----*/
if (prz->pSrcApp->SysType != ILTB_TYPE_APP)
{
    //----- Load communication module to connect with source system
    bLoadComm = TRUE;
    nSysType = prz->pSrcApp->SysType;
    nSysClass = prz->pSrcApp->SysClass;
    nComPort = prz->pSrcApp->ComPort;
}

//----- Do we need to connect to target handheld?
//----- (Note that Synchronization NEVER uses option==NONE)
else if ( (prz->pTarApp->SysType != ILTB_TYPE_APP)
    && (ILTR_nUpdOpt != ILX_OPT_NONE) )
{
    //----- Only handhelds having this attribute do reconciliation
    if (prz->pTarApp->SysAttrib & ILTB_ATT_SHOWNONE)
    {
        //----- Load communication module to connect with target system
        bLoadComm = TRUE;
        nSysType = prz->pTarApp->SysType;
        nSysClass = prz->pTarApp->SysClass;
        nComPort = prz->pTarApp->ComPort;
    }
}

/*-----
 * Load communication DLL only if necessary but only on the first
 * translation request.
 *-----*/
if (bLoadComm)
    //----- Load communication DLL and initiate connection
    nRc = ILX_LoadCommDLL (_ilx_globals, nSysType, nSysClass,
        nComPort, TRUE);

return nRc;

} //----- LoadCommIfNeeded

#endif // ILWIN

/*-----
 * CheckOrShowFieldMap (called from ILX_dTranslateEx)
 *-----*/
static int CheckOrShowFieldMap
(
    ILX_PGLOBALS _ilx_globals,
    ILX_PRECS prz,
    ILX_ID nMapId,
    ILX_BOOL bShowMap )
{
    int nRc;
    ILX_ACCESS nAccess;                // Access type
    ILX_BOOL bForceMap = ILX_FALSE;    // Force field re-mapping?
    ILX_MAP nMapType;                  // Type of mapping request

    //----- Source and target field lists
    int nSrcList = 0;                  // Number of source fields
    int nTarList = 0;                  // Number of target fields
    IL_HANDLE hSrcList = NULL;         // Handle to source field list
    IL_HANDLE hTarList = NULL;         // Handle to target field list

    //----- Has the user asked to see list of fields?
    if (!bShowMap)
    {

```

```

/*-----
 * Is the ILTB_ATT_SEC_USESRC section attribute set? If so, this
 * affects the nature of the call to ILX_dGetFieldMapEx. When the
 * option is set, it indicates that the target translator should
 * always be called to construct its field list before proceeding
 * with translation. This option is used in conjunction with the
 * section attribute ILTB_ATT_ASKFIELDS to force the field list
 * and associated mappings to match exactly those of the source
 * application. This logic only applies to non-existing data files.
 * The semantic result of this call is to make the target field
 * list the same as the source field list (with exact mappings).
 * The field map is retrieved, and immediately saved with modified
 * mappings to reflect the source field list.
 *-----*/

//----- Should we force target translator to remap fields?
if (prz->pTarSec->SectionAttrib & ILTB_ATT_SEC_USESRC)
{
    //----- Does the application support files?
    nAccess = prz->pTarSec->AccessType;
    if (nAccess != ILX_ACCESS_DDE && nAccess != ILX_ACCESS_NONE)
    {
        //----- Does the target file already exist?
        if (IL_DOESNT_EXIST (ILXGL_szTargetFile))
            bForceMap = ILX_TRUE;
    }
}

//----- Set the type of map request
nMapType = bForceMap ? ILX_MAP_LAST : ILX_MAP_VERIFY;

/*-----
 * Verify that at least one field is mapped in field lists.
 * We call the same function used to retrieve the field lists,
 * except that the last parameter is set to TRUE to indicate
 * that we only wish to validate the field map. In this case,
 * the output parameters are not really used and the field lists
 * are de-allocated in ILX_GetFieldMap before returning. If no
 * fields are mapped, the routine returns unique error code
 * ILX_ERR_NOMAPPING to allow the caller to display an error.
 *-----*/
nRc = ILX_dGetFieldMapEx ( _ilx_globals,
                          nMapId,
                          &hSrcList,
                          &hTarList,
                          &nSrcList,
                          &nTarList,
                          nMapType );

//----- Is at least one field mapped?
if (nRc == ILX_ERR_NOMAPPING)
{
    //----- If running on Macintosh, simply return the error code
    #ifdef ILMAC
        return ILX_ERR_NOMAPPING;
    #endif

    //----- If FORCE_FIELD MAPPING defined, Show error and force mapping
    #ifdef FORCE_FIELD_MAPPING
    {
        char szCaption [ILTB_MAX_MSG]; // Window caption
        char szText [ILTB_MAX_MSG]; // Message text
        ILST_HNDL hRes; // Resource file handle

        hRes = hDLLInstance;
        ILSTMakeString (&hRes, ILX_MSG_ERROR, szCaption, ILTB_MAX_MSG);
        ILSTMakeString (&hRes, ILX_MSG_NOMAP, szText, ILTB_MAX_MSG);

        //----- Display error message
        MessageBox ( ILXGL_hWindow,
                     (LPSTR) szText,
                     (LPSTR) szCaption,
                     MB_OK | MB_ICONEXCLAMATION | MB_TASKMODAL );

        //----- Force field mapping if no fields are mapped
    }
    #endif
}

```

```

        bShowMap = TRUE;
    }

    //----- If FORCE_FIELD_MAPPING not defined, simple return error code
    #else
        return ILX_ERR_NOMAPPING;
    #endif // FORCE_FIELD_MAPPING
}

else if (nRc)
    //----- Found an error
    return nRc;

else if (bForceMap)
{
    /*-----
    * If we succeeded in fetching the field lists from translators,
    * we proceed to update the field map at this point. This will
    * cause an exact mapping to be used in the translation step.
    *-----*/
    nRc = ILX_dPutFieldMapEx ( _ilx_globals,
                              nMapId,
                              hSrcList,
                              hTarList,
                              nSrcList,
                              nTarList );

    //----- Did we succeed in updating the field map?
    if (nRc)
        return nRc;
}

}

//----- Show the field map dialog if necessary
#ifndef WINPAD
    if (bShowMap)
    {
        nRc = ILX_dShowFieldMap (_ilx_globals);
        if (nRc)
            return nRc;
    }
#endif // WINPAD

return SUCCESS;
} //----- CheckOrShowFieldMap

/*-----
* Set_ILTR_Members (called from ILX_dTranslateEx)
*-----*/
static int Set_ILTR_Members
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr )
{
    //----- Transfer options to translation structure
    ILTR_version      = ILTR_CURRENT_VERSION;
    ILTR_nFunction     = prz->pSrcSec->SectionType;
    ILTR_log          = ILXGL_log;
    ILTR_nFilterID     = -1;
    ILTR_nTargetExportCharMapID = prz->pTarApp->ExportCharMapID;
    ILTR_nTargetImportCharMapID = prz->pTarApp->ImportCharMapID;
    ILTR_nSourceExportCharMapID = prz->pSrcApp->ExportCharMapID;
    ILTR_nSourceImportCharMapID = prz->pSrcApp->ImportCharMapID;
    IL_STRCPY (ILTR_szPswd, ILXGL_szPswd);
    IL_STRCPY (ILTR_szUserDir, ILXGL_szUserDir);

    //----- Put out all ASCII fields or just mapped ones?
    #if defined (WIN32) || defined (ILMAC)
        ILTR_CDFmapOnly = ILX_TRUE;
    #else
        ILTR_CDFmapOnly = ILX_FALSE;
    #endif
}

```



```

//---- next block is a backdoor convenience for testing...
{
    char szDate[20];
    GetProfileString ("ilwin", "TestLoDate", "", szDate, sizeof(szDate));
    if (IL_STRING_ISNT_NULL(szDate))
        ILXGL_lDateRangeStart = IL_AlphaToCodeDate (szDate);

    GetProfileString ("ilwin", "TestHiDate", "", szDate, sizeof(szDate));
    if (IL_STRING_ISNT_NULL(szDate))
        ILXGL_lDateRangeEnd = IL_AlphaToCodeDate (szDate);
}

/*-----
 * Set Date Range limits. Note that these limits may be adjusted in the
 * course of translation. TIF may adjust the date range differently for
 * each phase of Synchronization. Adjustments are influenced by the
 * System Types (ILTB_ATT_TOTAL_REBUILD) and the 'DEL_OUTRANGE' flags.
 *-----*/
ILTR_lDateRangeStart = ILXGL_lDateRangeStart; // tr->nLoDate
ILTR_lDateRangeEnd = ILXGL_lDateRangeEnd; // tr->nHiDate

/*-----
 * Set Date Range limits for Fanning. Unlike the other Date Range limits,
 * these settings should NEVER change over the course of a translation.
 *-----*/
ILTR_lFanningMinDate = ILXGL_lDateRangeStart;
ILTR_lFanningMaxDate = ILXGL_lDateRangeEnd;

ILTR_nSynchronize = ILXGL_nSynchronize;
ILTR_cbProgress = ILXGL_cbProgress;
ILTR_OKTP_Threshold = ILXGL_OKTP_Threshold; // OKToProceed threshold
ILTR_uTimerInterval = (UINT16) ILXGL_uTimerInterval;

//----- If running on Windows, prepare for WinHelp
#ifdef ILWIN
    ILTR_nHelpContext = ILXGL_nHelpOpen;
    ILTR_bVWRHelp = ILXGL_bVWRHelp;
    IL_STRCAT (IL_STRCPY (ILTR_szHelpFile, ILXGL_szDir), IL_FILESEP_STR);
    IL_STRCAT (ILTR_szHelpFile, ILXGL_szHelpFile);
#endif // ILWIN

//----- Convert environment to ILTR type:
switch (ILXGL_nEnviron)
{
    case ILX_ENV_NORMAL:
        ILTR_eEnvironment = ILTR_ENV_LITE;
        break;
    case ILX_ENV_LITE:
    case ILX_ENV_ACHATES:
        ILTR_eEnvironment = ILTR_ENV_LITE;
        break;
    case ILX_ENV_MAGICXCHANGE:
        ILTR_eEnvironment = ILTR_ENV_MAGICXCHANGE;
        break;
    case ILX_ENV_WINPAD:
        ILTR_eEnvironment = ILTR_ENV_WINPAD;
        break;

    default:
        //---- unknown environment type
        return ILERROR (ILXGL_nEnviron, ILX_ERR_INTERNAL);
}

/*-----
 * Place window handle, working directory name, instance handle,
 * and session ID (when relevant) in translation structure.
 *-----*/
IL_STRCPY (ILTR_szCurWD, ILXGL_szDir);

//----- Place Window and instance handles in translation structure
ILTR_hParentWin = ILXGL_hWindow;
ILTR_hParentInst = ILXGL_hInstance;
ILTR_hSessionID = ILXGL_hSessionID;

```

```

//----- Set appointment and todo range
if (ILTR_nFunction == ILTB_SEC_APPT)
{
    /*-----
    * Set the value of appointment range. If set to ILX_RANGE_DEFAULT,
    * we pull setting from the actual system record.
    *-----*/
    if (ILXGL_apptRange == ILX_RANGE_DEFAULT)
        ILTR_nSchOpt = prz->pSrcApp->ApptRange;
    else
        ILTR_nSchOpt = ILXGL_apptRange;
}
else if (ILTR_nFunction == ILTB_SEC_TODO)
{
    /*-----
    * Set the value of todo range. If set to ILX_RANGE_DEFAULT, we
    * pull the actual setting from the system record.
    *-----*/
    if (ILXGL_todoRange == ILX_RANGE_DEFAULT)
        ILTR_nSchOpt = prz->pSrcApp->TodoRange;
    else
        ILTR_nSchOpt = ILXGL_todoRange;
}

return SUCCESS;
} //----- Set_ILTR_Members

/*-----
* Set_trFlags (called from ILX_dTranslateEx)
*-----*/
static UINT32 Set_trFlags
( ILX_PGLOBS _ilx_globals,
  ILX_PRECS prz )
{
    char szFlags[20];
    UINT32 trFlags;

    //--- as a debugging convenience, allow a back door way to set
    //--- trFlags word. (see ILTR_FLAG_xxx definitions in ILTR.H)
    GetProfileString("ilwin", "ILTRflags", "0", szFlags, sizeof(szFlags));
    trFlags = (UINT32) strtoul ((const char *) szFlags, NULL, 10);

    if (ILXGL_keepFiles)
        trFlags |= ILTR_FLAG_KEEPPFILES;

    //--- set Macintosh flag if built for Macintosh
    #ifdef ILMAC
        trFlags |= ILTR_FLAG_MACINTOSH;
    #endif

    //--- set MIXED_WIN3216 flag if engine is 32-bit and either xlator is 16-bit
    #ifdef WIN32
        if ( ((prz->pSrcApp->SysAttrib & ILTB_ATT_ILXWIN_32) == 0)
            || ((prz->pTarApp->SysAttrib & ILTB_ATT_ILXWIN_32) == 0) )
            trFlags |= ILTR_FLAG_MIXED_WIN3216;
    #endif

    //--- set ILTR Source & Target 'DEL_OUTRANGE' flags
    if (prz->pSrcSec->SectionAttrib & ILTB_ATT_DEL_OUTRANGE)
        trFlags |= ILTR_FLAG_SOURCE_DEL_OUTRANGE;

    if (prz->pTarSec->SectionAttrib & ILTB_ATT_DEL_OUTRANGE)
        trFlags |= ILTR_FLAG_TARGET_DEL_OUTRANGE;

    //----- Is tagging and filtering disabled?
    if (ILXGL_Flags & ILX_FLAG_DISABLE_SST_TAGGING)
        trFlags |= ILTR_DISABLE_SST_TAGGING;
    if (ILXGL_Flags & ILX_FLAG_DISABLE_SST_FILTERS)
        trFlags |= ILTR_DISABLE_SST_FILTERING;

    //----- Are we operating in "unattended" mode?
    if (ILXGL_Flags & ILX_FLAG_UNATTENDED_MODE)
        trFlags |= ILTR_FLAG_UNATTENDED_MODE;
}

```

```

//----- Do we need to call the Chooser on Import?
if (ILXGL_Flags & ILX_FLAG_IMPORT_SELECTED)
    trFlags |= ILTR_FLAG_IMPORT_SELECTED;

return trFlags;
} //----- Set_trFlags

/*-----
 * StartLogFile (called from ILX_dTranslateEx)
 *-----*/
static int StartLogFile
( ILX_PGLOBS _ilx_globals,
  ILTR_PTRANSL tr,
  IL_PSTR pLogName )
{
    int nRc;
    char szText[ILTB_MAX_MSG];          // Message text
    IL_FILEINFO sFileInfo;              // used by IL_OPEN

    //----- Set default log file name if none was provided
    if (pLogName == NULL || pLogName[0] == '\0')
    {
        //----- Construct the name of log file
        IL_STRCPY (ILTR_szLogFile, ILXGL_szDir);
        if (ILTR_szLogFile[IL_STRLEN (ILTR_szLogFile)-1] != IL_FILESEP_CH)
            IL_STRCAT (ILTR_szLogFile, IL_FILESEP_STR);
        ILSTMakeString (&hDLLInstance, ILX_STR_XLATELOG, szText, ILTB_MAX_MSG);
        IL_STRCAT (ILTR_szLogFile, szText);
    }

    //----- Use supplied log file name
    else
        IL_STRCPY (ILTR_szLogFile, pLogName);

    /*-----
     * Header information is added to the log file by this routine
     * only when the log flag is set to TRUE.
     *-----*/
    if (ILXGL_log == ILX_TRUE && ILXGL_action == ILX_ACT_TRANSLATE)
    {
        int rc2;
        int openOption = IL_ATTR_WRITE;
        IL_HFILE hLog;          // Log file handle

        //----- Append to log file if not first translation
        if ( (ILXGL_Flags & ILX_FLAG_APPEND_TO_LOGS)
            && IL_DOES_EXIST(ILTR_szLogFile) )
            openOption = IL_ATTR_APPEND;

        //----- Create or overwrite or append to log file
        IL_OPEN (ILTR_szLogFile, openOption, hLog, sFileInfo, nRc);
        if (nRc)
            return ILX_ERR_LOG;

        //----- Write out header information to log file
        nRc = ILX_PutLogHeader (_ilx_globals, ILX_ACT_TRANSLATE, ILTR_nUpdOpt, &hLog);

        //----- Close log file at this point
        IL_CLOSE (hLog, rc2);

        //----- if ILX_PutLogHeader failed, exit now.
        if (nRc)
            return ILX_ERR_LOG;
    }

    return SUCCESS;
} //----- StartLogFile

#ifdef ILWIN

```

```

/*-----
 * InitProgressDisplay (called from ILX_dTranslateEx)
 *-----*/
static int InitProgressDisplay
( ILX_PGLOBALS _ilx_globals,
  ILTR_PTRANSL tr )
{
    int nRc;

    //----- Is this the first translation in session?
    if (ILXGL_bFirstXlate)
    {
        //----- Create and show the progress dialog
        nRc = ILX_ShowStatusMeter (tr, _ilx_globals);
        if (nRc != SUCCESS)
            return nRc;

        //----- Remember the status meter handle for subsequent translations
        ILXGL_hProgWin = ILTR_hProgWin;
    }
    else if (ILXGL_hProgWin)
    {
        //----- Reset handles to existing status dialog and controls
        ILTR_hProgWin = ILXGL_hProgWin;
        ILTR_hFromBarWin = GetDlgItem (ILXGL_hProgWin, ILCT_PRG_FROMBAR);
        ILTR_hToBarWin = GetDlgItem (ILXGL_hProgWin, ILCT_PRG_TOBAR);
    }

    return SUCCESS;
} //---- InitProgressDisplay

#endif // ILWIN

/*-----
 * PerformTranslations (called from ILX_dTranslateEx)
 *-----*/
static int PerformTranslations
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  UINT32 trFlags,
  int EnvAttribs )
{
    int nRc;

    /*-----
     * Translating APPT may involve two distinct translations
     * to process appointments and todo items separately. This
     * behavior is also affected by the setting of the range
     * options which may be set to ALL, FUTURE, or NONE.
     *-----*/
    if (ILTR_nFunction == ILTB_SEC_APPT)
    {
        //----- Is appointment range set to NONE?
        if (ILTR_nSchOpt != ILX_RANGE_NONE)
        {
            //----- Translate appointments
            nRc = doTranslate (_ilx_globals, prz, tr, trFlags, EnvAttribs);
            if (nRc != SUCCESS)
                return nRc;
        }

        /*-----
         * Now determine whether the TODO translator must also be run.
         * If neither Appt nor Todo range is set to NONE, and if BOTH systems
         * have the "ILX_ATT_TODO" system attribute, then we do it.
         *-----*/
        if ( (ILXGL_todoRange != ILX_RANGE_NONE)
            && (ILTR_nSchOpt != ILX_RANGE_NONE)
            && (prz->pSrcApp->SysAttrib & ILX_ATT_TODO)
            && (prz->pTarApp->SysAttrib & ILX_ATT_TODO) )
        {
            //----- Reset the function and range values

```

```

    ILTR_nFunction = ILTB_SEC_TODO;
    ILTR_nSchOpt = ILXGL_todoRange;

    //----- Translate Todo items
    nRc = doTranslate (_ilx_globals, prz, tr, trFlags, EnvAttribs);
    if (nRc != SUCCESS)
        return nRc;
    }
}

/*-----
 * Translating PHONE may involve two distinct translations for
 * PHONE and GROUPS data. The GROUPS translation is done if
 * BOTH systems have the 'ILX_ATT_GROUPS' attribute.
 *-----*/
else if (ILTR_nFunction == ILTB_SEC_PHONE)
{
    //----- Start by translating normal PHONE function
    nRc = doTranslate (_ilx_globals, prz, tr, trFlags, EnvAttribs);
    if (nRc != SUCCESS)
        return nRc;

    //----- Now determine whether GROUPS translator must be called
    if ( (prz->pSrcApp->SysAttrib & ILX_ATT_GROUPS)
        && (prz->pTarApp->SysAttrib & ILX_ATT_GROUPS) )
    {
        char szSourceSect[MAX_APP_NAME];
        char szTargetSect[MAX_APP_NAME];

        //----- make copies of "real" source and target section names
        IL_STRCPY (szSourceSect, prz->pSrcSec->Name);
        IL_STRCPY (szTargetSect, prz->pTarSec->Name);

        //----- Reset the section names to correspond to GROUPS
        ILSTMakeString ( &hDLLInstance, ILX_MSG_GRPSECT,
                        prz->pSrcSec->Name, MAX_APP_NAME );
        IL_STRCPY (prz->pTarSec->Name, prz->pSrcSec->Name);

        //----- Translate GROUPS data
        ILTR_nFunction = ILTB_SEC_GROUPS;
        nRc = doTranslate (_ilx_globals, prz, tr, trFlags, EnvAttribs);

        //----- restore "real" source and target section names
        IL_STRCPY (prz->pSrcSec->Name, szSourceSect);
        IL_STRCPY (prz->pTarSec->Name, szTargetSect);

        if (nRc != SUCCESS)
            return nRc;
    }
} //----- if ... ILTB_SEC_APPT ... else if ... ILTB_SEC_PHONE ...

else
{
    //----- Perform single translation
    nRc = doTranslate (_ilx_globals, prz, tr, trFlags, EnvAttribs);
    if (nRc != SUCCESS)
        return nRc;
}

return SUCCESS;
} //----- PerformTranslations

/*-----
 * Phase Definitions, used when ILX is operating in "ILX_V4" mode,
 * i.e. "Version 4", where ILTIF is the center of the universe.
 *
 * ILTR_PHASE01: initial settings
 * ILTR_PHASE05: "pre-export", from Source -- for ILX_V4 only
 * ILTR_PHASE10: "backward" export, from Target -- for ILX_V4 only
 * ILTR_PHASE20: normal export, from Source
 * ILTR_PHASE30: normal import, into Target
 * ILTR_PHASE40: "backward" import, into Source -- for SYNC only

```

```

*
* Origin of key parameter settings, per phase:
*
*      (S for Source, T for Target)
*
* Parameter      Phase01  Phase05  Phase10  Phase20  Phase30  Phase40
* -----
* ILTR_nCmd       EXPORT  EXPORT  EXPORT  EXPORT  IMPORT  IMPORT
* ILTR_direction  1      1      1      1      0      0
* ILTR_nSourceID   S      S      T      S      S      T
* ILTR_nSrcSection S      S      T      S      S      T
* ILTR_nTargetID   T      T      S      T      T      S
* ILTR_nTarSection T      T      S      T      T      S
* ILTR_nSysClass   -      T      T      S      T      S
* ILTR_nSysType    -      T      T      S      T      S
* ILTR_nAttribs    env    T      T      S      T      S
* ILTR_nSectionAttribs -    T      T      S      T      S
* ILTR_szAppName  -      T      T      S      T      S
* ILTR_szSectCode  -      T      T      S      T      S
* ILTR_szSectName  -      T      T      S      T      S
* ILTR_szAppFile   -      T      T      S      T      S
* ILTR_szAltApp    -      S      S      T      S      T
* ILTR_szAltSect   -      S      S      T      S      T
* ILTR_pXtraData   -      T      T      S      T      S
* ILTR_szSrcTrans  S      S      T      S      S      T
* ILTR_szTarTrans  T      T      S      T      T      S
* ILTR_nSubSectionType -    T      T      S      T      S
* ILTR_SourceSST   S      S      T      S      S      T
* ILTR_TargetSST   T      T      S      T      T      S
* ILTR_pTargetSSTS -      T      S      T      T      S
*
* ----- the following params are set by iltr\loadmap.c\ILSetupTables,
* ----- which is called by ILX_TellTIFAboutFields and by
* ----- iltr\import.c and iltr\export.c
* ILTR_map.pSource -      S      T      S      S      T
* ILTR_map.pTarget -      T      S      T      T      S
* ILTR_sCharMap    -      S      T      S      T      S
*
* -----
* NOTE: Phase 05 settings are mostly immaterial. The dominant consideration
* for Phase 05 settings is to imitate Phase 30 settings. This is because
* the first time initialization of ILTIF always takes place in Phase 05 for
* ILX_V4 operation, or Phase 30 for ILX_V3 operation. We want TIF to
* start up in as similar a setting as possible for ILX_V3 and ILX_V4.
* -----*/
/*-----
* Name: SetPhaseParams
* Called from doTranslate
*-----*/
static int SetPhaseParams ( ILX_PGLOBALS _ilx_globals,
                           ILX_PRECS prz,
                           ILTR_PTRANSL tr,
                           int EnvironmentAttribs,
                           char phase )
{
    /*-----
    * For ILX_V4 mode only, put phase# into ILTR_phase.
    * NOTE: for ILX_V3 mode, ILTR_phase remains ZERO throughout.
    *-----*/
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
        ILTR_phase = phase;

    /*-----
    * Start every phase with the ILTR_FLAG_SEE_IF_FILE_EXISTS flag cleared.
    * This function is only ever set in export phases (10 & 20), and then
    * only where it makes sense to check for existence of the app data file.
    *-----*/
    ILTR_Flags &= ~ILTR_FLAG_SEE_IF_FILE_EXISTS;

    /*-----
    * Set constant default parameters. These values may be altered when a
    * translator runs. At the start of each phase we set them back to their
    * default values so that each xlator run gets a clean set of default
    * values, regardless of what the previous xlator may have done.
    *-----*/
    ILTR_CDFsep = ILXGL_AsciiSep; // ASCII delimiter to use

```

```

ILTR_CDFnames = ILXGL_AsciiNames;      // Are field names in 1st record?

/*-----
 * The following constant parameters are used to restrict fanning of
 * recurring items. See comments in ILTR.H for further explanation.
 * Translator's can adjust these counts when 'OpenDataStore' is called.
 * When translating between a pair of apps, the translators for each app
 * can apply whatever limits it likes; they need not agree on values.
 * The values assigned here are not sacred in any way; they're just
 * a stab at being reasonable...
 *
 * Change made 9/18/96 by DJB for Starfish (Sidekick) to make fanning of
 * Weekly and Monthly patterns from Schedule+ 7.0 work better. Previously
 * weekly=-52, monthly=-12. Now upped both to allow for patterns that
 * start and stop on the same day of the year.
 *-----*/
ILTR_nDailyMaxFanout      = -31;      // max for DAILY patterns
ILTR_nWeeklyMaxFanout     = -53;      // max for WEEKLY & WEEKLY_DAYS
ILTR_nMonthlyMaxFanout    = -13;      // max for all 3 MONTHLY patterns
ILTR_nQuarterlyMaxFanout  = -8;       // max for QUARTERLY patterns
ILTR_nYearlyMaxFanout     = -5;       // max for all 3 YEARLY patterns
ILTR_nOtherMaxFanout      = 50;       // max for all other patterns

/*-----
 * Set direction parameters (1-5-10-20-30-40 pattern is EEEEEII)
 *-----*/
if (phase < ILTR_PHASE30)
{
    ILTR_nCmd = ILX_IO_EXPORT;
    ILTR_direction = ILTR_EXPORT;
}
else
{
    ILTR_nCmd = ILX_IO_IMPORT;
    ILTR_direction = ILTR_IMPORT;
}

/*-----
 * Set SOURCE parameters (1-5-10-20-30-40 pattern is SSTSST)
 *-----*/
if (phase == ILTR_PHASE10 || phase == ILTR_PHASE40)
{
    //---- backward settings
    ILTR_nSourceID = prz->pTarApp->SysID;
    ILTR_nSrcSection = prz->pTarSec->SectionID;
    IL_STRCPY (ILTR_szSrcTrans, prz->pTarApp->XlateName);
    ILTR_SourceSST = (BYTE) prz->pTarSec->SectionSubType;
    ILTR_hAppSession = ILXGL_hTarAppSession;
}
else
{
    //---- forward settings
    ILTR_nSourceID = prz->pSrcApp->SysID;
    ILTR_nSrcSection = prz->pSrcSec->SectionID;
    IL_STRCPY (ILTR_szSrcTrans, prz->pSrcApp->XlateName);
    ILTR_SourceSST = (BYTE) prz->pSrcSec->SectionSubType;
    ILTR_hAppSession = ILXGL_hSrcAppSession;
}

/*-----
 * Set TARGET parameters (1-5-10-20-30-40 pattern is TTSTTS)
 *-----*/
if (phase == ILTR_PHASE10 || phase == ILTR_PHASE40)
{
    //---- backward settings
    ILTR_nTargetID = prz->pSrcApp->SysID;
    ILTR_nTarSection = prz->pSrcSec->SectionID;
    IL_STRCPY (ILTR_szTarTrans, prz->pSrcApp->XlateName);
    ILTR_TargetSST = (BYTE) prz->pSrcSec->SectionSubType;
    ILTR_pTargetSSTS = &ILTR_pTableInfo->sOriginalSourceSSTList;
    ILTR_hAppSession = ILXGL_hSrcAppSession;
}
else
{
    //---- forward settings

```



```

    ILTR_nTargetID = prz->pTarApp->SysID;
    ILTR_nTarSection = prz->pTarSec->SectionID;
    IL_STRCPY (ILTR_szTarTrans, prz->pTarApp->XlateName);
    ILTR_TargetSST = (BYTE) prz->pTarSec->SectionSubType;
    ILTR_hAppSession = ILXGL_hTarAppSession;
    if (phase != ILTR_PHASE01)
        ILTR_pTarGetSSTS = &ILTR_pTableInfo->sOriginalTargetSSTList;
}

/*-----
 * For phase01 set one more parameter, then bail out
 *-----*/
if (phase == ILTR_PHASE01)
{
    ILTR_nAttribs = EnvironmentAttribs;
    return SUCCESS;
}

/*-----
 * Set ACTIVE TRANSLATOR parameters (5-10-20-30-40 pattern is TTSTS)
 *-----*/
if (phase == ILTR_PHASE20 || phase == ILTR_PHASE40)
{
    /*---- SOURCE translator is active
    ILTR_nSysClass = prz->pSrcApp->SysClass;
    ILTR_nSysType = prz->pSrcApp->SysType;
    ILTR_nAttribs = prz->pSrcApp->SysAttrib | EnvironmentAttribs;
    ILTR_nSectionAttribs = prz->pSrcSec->SectionAttrib;
    IL_STRCPY (ILTR_szAppName, prz->pSrcApp->SysName);
    IL_STRCPY (ILTR_szSectCode, prz->pSrcSec->Code);
    IL_STRCPY (ILTR_szSectName, prz->pSrcSec->Name);
    IL_STRCPY (ILTR_szAppFile, ILXGL_szSourceFile1);
    ILTR_pXtraData = ILXGL_pSourceXtraData;
    ILTR_nSubSectionType = prz->pSrcSec->SectionSubType;
    }
    else
    {
        /*---- TARGET translator is active
        ILTR_nSysClass = prz->pTarApp->SysClass;
        ILTR_nSysType = prz->pTarApp->SysType;
        ILTR_nAttribs = prz->pTarApp->SysAttrib | EnvironmentAttribs;
        ILTR_nSectionAttribs = prz->pTarSec->SectionAttrib;
        IL_STRCPY (ILTR_szAppName, prz->pTarApp->SysName);
        IL_STRCPY (ILTR_szSectCode, prz->pTarSec->Code);
        IL_STRCPY (ILTR_szSectName, prz->pTarSec->Name);
        IL_STRCPY (ILTR_szAppFile, ILXGL_szTargetFile);
        ILTR_pXtraData = ILXGL_pTargetXtraData;
        ILTR_nSubSectionType = prz->pTarSec->SectionSubType;
    }

    /*-----
    * Set OTHER (Alt) TRANSLATOR parameters (5-10-20-30-40 pattern is TTSTS)
    *-----*/
    if (phase == ILTR_PHASE20 || phase == ILTR_PHASE40)
    {
        /*---- SOURCE xlator is active, so TARGET xlator is "Alt"
        IL_STRCPY (ILTR_szAltSect, prz->pTarSec->Name);
        IL_STRCPY (ILTR_szAltApp, prz->pTarApp->SysName);
        }
        else
        {
            /*---- TARGET xlator is active, so SOURCE xlator is "Alt"
            IL_STRCPY (ILTR_szAltSect, prz->pSrcSec->Name);
            IL_STRCPY (ILTR_szAltApp, prz->pSrcApp->SysName);
        }

    /*-----
    * Tweak ILXGL parameters for handhelds, if 'Use Table' is set
    *-----*/
    if (ILTR_nSysType != ILTB_TYPE_APP)
    {
        /*-----
        * ACTIVE system is a device (therefore requiring communication).
        * Override the RAM and Document options when the caller has
        * specified that system table options should take precedence.

```

```

    /*-----*/
    if (ILXGL_bUseTable)
    {
        ILXGL_bRamCard = (ILTR_nAttribs & ILTB_ATT_RAMCARD) ? 1 : 0;
        ILXGL_bDocument = (ILTR_nAttribs & ILTB_ATT_DOCNAME) ? 1 : 0;
    }
}

/*-----
 * Augment Section Attributes for ACTIVE system
 *-----*/
if (!ILXGL_bFanRepeat)
    ILTR_nSectionAttribs |= ILTB_ATT_NOFANNING;
if (ILXGL_bRamCard)
    ILTR_nSectionAttribs |= ILTB_ATT_PCCARD_RAM;

return SUCCESS;
}

/*-----
 * If built with "ILTIF_IN_ENGINE" defined, the ILX DLL can't live without
 * the ILTIF DLL.
 *-----*/
#ifdef ILTIF_IN_ENGINE

/*-----
 * Name:  MapTIFReturnCode
 *-----*/
static int MapTIFReturnCode (ILTR_PTRANSL tr, int tifrc)
{
    if (ILTR_rc == SUCCESS)
        ILTR_rc = (INT16) tifrc;

    //----- use standard ILTR-to-ILX error code mapping
    return ILX_MapILTRErrorCode (tifrc);
}

/*-----
 * Name:  ILX_ILTIFInit
 *-----*/
static int ILX_ILTIFInit ( ILTR_PTRANSL tr,
                          IL_PSTR lpszTIFFilename,
                          INT32 lFields )
{
    int rc = ILTIFInit(tr, lpszTIFFilename, lFields);
    rc = MapTIFReturnCode(tr, rc);
    return rc;
}

/*-----
 * Name:  ILX_ILTIFClose
 *-----*/
static int ILX_ILTIFClose (ILTR_PTRANSL tr)
{
    int rc = ILTIFClose(tr);
    rc = MapTIFReturnCode(tr, rc);
    return rc;
}

/*-----
 * Name:  ILX_ILTIFSyncInit
 *-----*/
static int ILX_ILTIFSyncInit ( ILTR_PTRANSL tr,
                              IL_PSTR szSourceFile,
                              IL_PSTR szTargetFile )
{
    int rc = ILTIFSyncInit(tr, szSourceFile, szTargetFile);
    rc = MapTIFReturnCode(tr, rc);
    return rc;
}

```

```

/*-----
 * Name:  ILX_ILTIFSyncFinishUpAndClose
 *-----*/
static int ILX_ILTIFSyncFinishUpAndClose (ILTR_PTRANSL tr)
{
    int rc = ILTIFSyncFinishUpAndClose(tr);
    rc = MapTIFReturnCode(tr, rc);
    return rc;
}

/*-----
 * Name:  ILX_ILTIFStartNextPhase
 *-----*/
static int ILX_ILTIFStartNextPhase (ILTR_PTRANSL tr, INT16 phase)
{
    int rc = ILTIFStartNextPhase(tr, phase);
    rc = MapTIFReturnCode(tr, rc);
    return rc;
}

/*-----
 * Name:  TellTIFAboutFields -- called from within doTranslate
 *-----*/
static int TellTIFAboutFields (ILTR_PTRANSL tr)
{
    return ILX_TellTIFAboutFields (tr);
}

/*-----
 * Name:  ILX_ILTIFCloseFileInitially -- called from within doTranslate
 *-----*/
static int ILX_ILTIFCloseFileInitially (ILTR_PTRANSL tr)
{
    return ILTIFCloseFileInitially (tr);
}

/*-----
 * Name:  ILX_ILTIFReopenFile -- called from within doTranslate
 *-----*/
static int ILX_ILTIFReopenFile (ILTR_PTRANSL tr)
{
    return ILTIFReopenFile (tr);
}

#else
/*-----
 * If ILX isn't built with "ILTIF_IN_ENGINE", it is a boo-boo to
 * make calls to ILTIF entrypoints!!
 *-----*/
#define ILX_ILTIFInit(a,b,c)                ILX_ERR_CANT_USE_ILTIF
#define ILX_ILTIFClose(a)                   ILX_ERR_CANT_USE_ILTIF
#define ILX_ILTIFStartNextPhase(a,b)        ILX_ERR_CANT_USE_ILTIF

#define ILX_ILTIFSyncInit(a,b,c)             ILX_ERR_CANT_USE_ILTIF
#define ILX_ILTIFSyncFinishUpAndClose(a)    ILX_ERR_CANT_USE_ILTIF

//----- plus ILX_V4 local functions (for now)
#define TellTIFAboutFields(a)                ILX_ERR_CANT_USE_ILTIF

/*-----
 * If ILX isn't built with "ILTIF_IN_ENGINE", the following TIF
 * calls are simply ignored.
 *-----*/
#define ILX_ILTIFCloseFileInitially(a)       SUCCESS
#define ILX_ILTIFReopenFile(a)              SUCCESS
#endif // #ifdef ILTIF_IN_ENGINE ... #else

```

```

/*-----
* Name:      doTranslate (called from PerformTranslations)
*
* Purpose:   Perform complete set of export and import operations. For
*            simple (import/export) translation, an export is performed
*            for the source and an import for the target. For SyncPort,
*            we export twice, first from the target and then from the
*            source, and then import twice, first to the target and then
*            to the source.
*
* Parameters:
*   _ilx_globals - Pointer to global data
*   prz - Pointers to system and section records
*   tr - Pointer to translation structure
*   bFlag - Boolean flag
*   trFlags - base value for ILTR_Flags
*   EnvAttribs - environmental bits to OR into ILTR_nAttribs
*
* Returns:   ILX_OK or error code
*-----*/
static int doTranslate
(
    ILX_PGLOBS _ilx_globals,
    ILX_PRECS prz,
    ILTR_PTRANSL tr,
    UINT32 trFlags,
    int EnvAttribs )
{
    int nRc;
    int rc2;
    ILTB_PHNDL phTable = NULL;           // Pointer to table handle
    IL_FILEINFO sFileInfo;              // used by IL_REMOVE
    char szFile[_MAX_PATH];             // File name

    //---- set ILTR_Flags for this pass (NOTE: flags may be altered during run)
    ILTR_Flags = trFlags;

    //---- set or clear the ILTR First in Session Flag
    if (ILXGL_Flags & ILX_FLAG_FIRST_DOTRANSLATE)
        ILTR_Flags |= ILTR_FLAG_FIRST_XLATE;
    else
        ILTR_Flags &= ~ILTR_FLAG_FIRST_XLATE;

    /*-----
    * Set ILTR Append-to-Logs flag as needed. The only time this flag is
    * ever CLEAR is for the very first 'doTranslate' call in a session,
    * or for the very first 'doTranslate' in a multi-session job.
    *-----*/
    if (ILXGL_Flags & ILX_FLAG_APPEND_TO_LOGS)
        ILTR_Flags |= ILTR_FLAG_APPEND_TO_LOGS;

    //---- fiddle with ILTR_nSynchronize and initial ILTR_phase settings
    nRc = Choose_V3_V4_Or_Sync (prz, tr);
    if (nRc != SUCCESS)
        goto LastExit;

    //---- For sync, do 'sync from scratch' if file(s) to sync is/are new
    if (ILTR_nSynchronize == ILXTR_SYNC_STANDARD)
        SyncFromScratchIfFilesAreNew (_ilx_globals, prz, tr);

    /*-----
    * For both ILX_V3 and ILX_V4, set initial ILTR parameter values
    *-----*/
    nRc = SetPhaseParams (_ilx_globals, prz, tr, EnvAttribs, ILTR_PHASE01);
    if (nRc != SUCCESS)
        goto LastExit;

    //----- Load source/target field list and mapping tables into the tr record
    phTable = (ILTB_PHNDL) GlobalLock (ILXGL_hTable);
    nRc = ILX_LoadFieldLists (tr, phTable);
    if (nRc != SUCCESS)
    {
        ILXGL_nXlatorError = nRc;
        nRc = ILX_MapILTRErrorCode (nRc);
        goto LastExit;
    }
}

```

```

if (ILTR_pTableInfo == NULL)
{
    nRc = ILX_ERR_INTABLE;
    goto LastExit;
}

/*-----
 * Close the system table (TABLES.ITB) to prevent data loss.
 * WARNING: for all failures beyond this point, you must re-open
 * the system table. (use EXIT_WITH_ERROR macro or goto LaterExit).
 *-----*/
ILTBClose (phTable);

/*-----
 * If Operating in "ILX_V4" mode, Open or Create a TIF File
 *-----*/
if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
{
    nRc = do_Phase05_Setup (_ilx_globals, prz, tr, EnvAttribs);
    if (nRc != SUCCESS)
        goto LaterExit;
}

//----- Clear counters and flags
ILXGL_lTotalRecords = 0L;
ILXGL_nXlateDataErrs = 0;

//----- Clear temp filenames
IL_STRCPY (szTempRemoteSource, "");
IL_STRCPY (szTempRemoteTarget, "");

//----- If using ILIF, get an intermediate file name
if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
{
    IL_PSTR psz = ILTR_TempFileName (ILTR_nFunction, ILTR_szWorkFile);
    if (psz == NULL)
        EXIT_WITH_ERROR (ILX_NOTOK);
}

/*-----
 * If Operating in "ILX_V4" mode, Export from Target App before
 * Exporting from Source App. For "ILX_V4" TIF-based translation
 * we need to do this even if the Conflict Resolution Option is
 * set to NONE, cuz it's up to the Target Translator to establish
 * the bulk of the TIF Field List at this point.
 *-----*/
if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
{
    nRc = do_Phase10_ExportFromTarget (_ilx_globals, prz, tr, EnvAttribs);
    if (nRc != SUCCESS)
        EXIT_WITH_ERROR (nRc);
}

/*-----
 * For both ILX_V3 and ILX_V4 mode, Export from Source App.
 *-----*/
nRc = do_Phase20_ExportFromSource (_ilx_globals, prz, tr, EnvAttribs);
if (nRc != SUCCESS)
    EXIT_WITH_ERROR (nRc);

/*-----
 * For both ILX_V3 and ILX_V4 mode, Import into Target App.
 *-----*/
nRc = do_Phase30_ImportIntoTarget (_ilx_globals, prz, tr, EnvAttribs);
if (nRc != SUCCESS)
    EXIT_WITH_ERROR (nRc);

if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
{
    //----- this is an ILX_V4 job, so we have a bit more work to do...

    //----- If we're not doing Synchronization, finish up now.
    if (ILTR_nSynchronize == ILXTR_SYNC_NO)
    {
        //----- for SmartMerge the ILTIFClose call deletes the TIF file
    }
}

```

```

        //----- NOTE: doesn't matter whether TIF workfile is open or not
        nRc = ILX_ILTIFClose(tr);
        if (nRc != SUCCESS)
            EXIT_WITH_ERROR (nRc);
    }

    else
    {
        /*-----
        * For Synchronization, Import into Source App.
        *-----*/
        nRc = do_Phase40_ImportIntoSource (_ilx_globals, prz, tr, EnvAttribs);
        if (nRc != SUCCESS)
            EXIT_WITH_ERROR (nRc);

        /*-----
        * Then Create New Sync History File and finish up.
        *-----*/
        nRc = do_SyncFinishUp (tr);
        if (nRc != SUCCESS)
            EXIT_WITH_ERROR (nRc);
    }

    } //---- if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)

Exit:

//---- Remove intermediate file if not using TIF and Keep Files not set.
if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
{
    if (ILXGL_keepFiles == ILX_FALSE)
        IL_REMOVE (ILTR_szWorkFile, sFileInfo, rc2);
}

//----- If using TIF and we have an error, make sure TIF is shut down
else if (nRc != SUCCESS)
{
    //--- TIF workfile is deleted if KeepFiles isn't set.
    rc2 = ILX_ILTIFClose (tr);        // rc2 ignored!!
}

//----- Remember the last password used
IL_STRCPY (ILXGL_szPswd, ILTR_szPswd);

LaterExit:

/*-----
* record ILTR error codes in ILXGL structure. This is probably
* only necessary when direct ILX-to-TIF calls return error.
*-----*/
if (ILXGL_nXlatorError == 0)
    ILXGL_nXlatorError = ILTR_rc;

if (ILXGL_nSystemError == 0)
    ILXGL_nSystemError = ILTR_nSystemError;

//----- Free any allocated field list and mapping tables in the tr record
ILX_FreeFieldLists (tr);

//----- Reopen the table
IL_STRCPY (szFile, ILXGL_szDir);
if (szFile[IL_STRLEN (szFile)-1] != IL_FILESEP_CH)
    IL_STRCAT (szFile, IL_FILESEP_STR);
IL_STRCAT (szFile, ILX_TABLES_FILE);

/*-----
* Open System Table File in UPDATE mode, except that if the file is
* READONLY, open it in READ mode. Most of our apps, including
* IntelliLink Lite, need UPDATE access, but some, e.g. IntelliLink
* for WinPad, need nothing more than READ access.
*-----*/
if (ILTBOpen (szFile, phTable, ILTB_MODE_AS_ALLOWED) != ILTB_OK)
{
    if (nRc == SUCCESS)

```

```

        //----- no previous errors to report, so let's report this one
        nRc = ILX_ERR_INTABLE;
    }

LastExit:

    //----- Any subsequent doTranslate calls in same session are NOT 1st
    ILXGL_Flags |= ILX_FLAG_APPEND_TO_LOGS;

    //----- Reset the "first translation" flag unless we need to "re-sync"
    if (nRc != ILX_ERR_RESYNC)
        ILXGL_Flags &= (~ILX_FLAG_FIRST_DOTRANSLATE);

    return (nRc);
} //---- doTranslate

/*-----
 * Choose_V3_V4_Or_Sync (called from doTranslate)
 *-----*/
static int Choose_V3_V4_Or_Sync
( ILX_PRECS prz,
  ILTR_PTRANSL tr )
{
    /*-----
     * Comments on setting the ILTR_phase and ILTR_nSynchronize flags:
     *
     * ILTR_phase is used to choose between ILX_V3 and ILX_V4 styles of
     * operation:
     *
     * Zero selects ILX_V3 mode; anything nonzero selects ILX_V4 mode.
     *
     * The ILX_V3 style is the traditional style, where ILIF is
     * used as the intermediate file, and "doTranslate" simply calls SOURCE
     * translator EXPORT, then TARGET translator IMPORT.
     *
     * The ILX_V4 style is the new style, where ILTIF is used as the
     * intermediate file, and "doTranslate" orchestrates 3 or more phases
     * of translation -- export from target, export from source, import
     * into target...
     *
     * ILTR_nSynchronize is used to choose whether we're doing a one-way
     * SmartMerge operation or doing a two-way SYNCHRONIZATION job.
     *
     * There is also a special value for ILTR_nSynchronize,
     * ILXTR_SYNC_NO_BUT_USE_ILXV4, which allows an app to choose the ILX_V4
     * style of operation for a SmartMerge operation.
     *
     * So, a fully adapted app should use the following ILX_SetValue settings
     * for ILX_VAL_SYNCHRONIZE:
     *
     * To do Synchronization (which is always TIF-based): ILXTR_SYNC_STANDARD
     *
     * To do an ILIF-based SmartMerge: ILXTR_SYNC_NO
     *
     * To do a TIF-based SmartMerge: ILXTR_SYNC_NO_BUT_USE_ILXV4
     *-----*/

    switch (ILTR_nSynchronize)
    {
        case ILXTR_SYNC_NO:                //---- ILX_V3 SmartMerge

            ILTR_phase = ILTR_PHASE_ILX_V3_MODE;
            break;

        case ILXTR_SYNC_NO_BUT_USE_ILXV4:  //----- ILX_V4 SmartMerge

            /*-----
             * Use ILX_V4 mode if engine is built to support it and if
             * target translator is Sync Capable. Otherwise revert to
             * ILX_V3 mode.
             *-----*/
            #ifdef ILTIF_IN_ENGINE

```



```

        if (prz->pTarApp->SysAttrib & ILTB_ATT_SYNC_CAPABLE)
            ILTR_phase = ~ILTR_PHASE_ILX_V3_MODE;
        else

#endif

        ILTR_phase = ILTR_PHASE_ILX_V3_MODE;

    ILTR_nSynchronize = ILXTR_SYNC_NO; // NOT synchronizing

    break;

case ILXTR_SYNC_STANDARD:                //---- ILX_V4 Synchronization
case ILXTR_SYNC_FROM_SCRATCH:

    if ( ((prz->pSrcApp->SysAttrib & ILTB_ATT_SYNC_CAPABLE) == 0)
        || ((prz->pTarApp->SysAttrib & ILTB_ATT_SYNC_CAPABLE) == 0) )
        //---- cannot do sync unless BOTH xlaters are sync-capable
        return ILX_ERR_BAD_SYNC_OPTION;

    //----- for synchronization we always use ILX_V4 mode
    ILTR_phase = ~ILTR_PHASE_ILX_V3_MODE;
    break;

default:                                //---- BAD sync code
    return ILX_ERR_BAD_SYNC_OPTION;

} //---- switch (ILTR_nSynchronize)

return SUCCESS;

} //---- Choose_V3_V4_Or_Sync

/*-----
 * SyncFromScratchIfFilesAreNew (called from doTranslate)
 *
 * Do 'sync from scratch' if one or the other or both of the Files to be
 * synchronized is non-existent or zero-length at the start of the job.
 * This prevents us from doing a DELETE-ALL to the other sync partner.
 *
 * WARNING: this protection only applies to those DeskTop apps for which
 * we know a filename, and it doesn't guard against tricky scenarios
 * such as switching from one AddressBook section to another within the
 * same Lotus Organizer file.
 *-----*/
static void SyncFromScratchIfFilesAreNew
( ILX_PGLOBS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr )
{
    //---- Check to see if the Source File exists and is non-zero-length
    if (prz->pSrcApp->SysType == ILTB_TYPE_APP)
        switch (prz->pSrcApp->AccessType)
        {
            case ILX_ACCESS_DDE:                // Dynamic Data Exchange
            case ILX_ACCESS_ODBC:                // ODBC database
            case ILX_ACCESS_NEW1:                // Unassigned
            default:
                break;

            case ILX_ACCESS_DBASE:                // dBASE database
            case ILX_ACCESS_FILE:                // File access
            case ILX_ACCESS_PDX:                // Paradox database
            case ILX_ACCESS_CDF:                // Ascii file type

                if ( IL_DOESNT_EXIST(ILXGL_szSourceFile1)
                    || ILUT_IsZeroFile(ILXGL_szSourceFile1) )
                {
                    ILTR_nSynchronize = ILXTR_SYNC_FROM_SCRATCH;
                    return;
                }
            }
        }

    //---- Check to see if the Target File exists and is non-zero-length

```

```

if (prz->pTarApp->SysType == ILTB_TYPE_APP)
    switch (prz->pTarApp->AccessType)
    {
        case ILX_ACCESS_DDE:                // Dynamic Data Exchange
        case ILX_ACCESS_ODBC:               // ODBC database
        case ILX_ACCESS_NEW1:               // Unassigned
        default:-
            break;

        case ILX_ACCESS_DBASE:              // dBASE database
        case ILX_ACCESS_FILE:               // File access
        case ILX_ACCESS_PDX:                // Paradox database
        case ILX_ACCESS_CDF:                // Ascii file type

            if ( IL_DOESNT_EXIST(ILXGL_szTargetFile)
                || ILUT_IsZeroFile(ILXGL_szTargetFile) )
            {
                ILTR_nSynchronize = ILXTR_SYNC_FROM_SCRATCH;
                return;
            }
    }

} //---- SyncFromScratchIfFilesAreNew

/*-----
 * do_Phase05 (called from doTranslate)
 *-----*/
static int do_Phase05_Setup
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs )
{
    int nRc, rc2;

    /*-----
     * Set phase parameters for TIF init and pre-export from source
     *-----*/
    nRc = SetPhaseParams (_ilx_globals, prz, tr, EnvAttribs, ILTR_PHASE05);
    if (nRc != SUCCESS)
        return nRc;

    if (ILTR_nSynchronize == ILXTR_SYNC_NO)
        nRc = ILX_ILTIFInit (tr, NULL, 0);
    else
        nRc = ILX_ILTIFSyncInit (tr, ILXGL_szSourceFile1, ILXGL_szTargetFile);

    if (nRc != SUCCESS)
        return nRc;

    /*-----
     * Now ask the Source Translator to identify all its fields.
     * (set direction so we see SOURCE side of Field Map)
     *-----*/
    nRc = TellTIFAboutFields (tr);
    if (nRc != SUCCESS)
    {
        rc2 = ILX_ILTIFClose(tr); // rc2 is ignored!!
        return nRc;
    }

    return SUCCESS;
} //---- do_Phase05_Setup

/*-----
 * do_Phase10_ExportFromTarget (ILX_V4 only) (called from doTranslate)
 *
 * NOTE: both this function and the do_Phase20_ExportFromSource function
 * used to have shortcuts to bypass calling ILX_LoadTranslator when the
 * app data file doesn't exist. For all ILX_V4 operations these
 * shortcuts are now deferred to ILXTRANS, because TIF relies on being
 * called from EXPORT.C to squirrel away some phase-specific parameters.
 *-----*/

```

```

* (i.e. ILTIFReopenFile must be called).
* The ILTR_FLAG_SEE_IF_FILE_EXISTS flag now enables the deferred shortcut.
*-----*/
static int do_Phase10_ExportFromTarget
( ILX_PGLOBALS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs )
{
    int nRc;                // Return code
    int nTempRc;            // Temp return code
    IL_FILEINFO sFileInfo;  // used by IL_REMOVE

    /*-----
    *
    * Start of Step 1 -- "Backwards" Export from TARGET (sic!)
    *
    * This is how ILX_V4 does "export before import".
    *-----*/

    /*-----
    * Set phase parameters for "Backwards" Export from TARGET
    *-----*/
    nRc = SetPhaseParams (_ilx_globals, prz, tr, EnvAttribs, ILTR_PHASE10);
    if (nRc != SUCCESS)
        return nRc;

    /*----- Get Target Translator to tell TIF about its fields
    nRc = TellTIFAboutFields (tr);
    if (nRc != 0)
        return nRc;

    /*--- for Synchronization, assimilate previous history file
    if (ILTR_nSynchronize == ILXTR_SYNC_STANDARD)
    {
        nRc = ILX_ILTIFStartNextPhase(tr, TIF_PHASE_LOADING_PREVIOUS_RECORDS);
        if (nRc != 0)
            return nRc;
    }

    /*-----
    * Always cycle TIF through the 'LoadingTargetRecords' phase, whether or
    * not there are any records to load. This ensures that TIF will get a
    * chance to save any parameters associated with this phase (e.g.
    * Loading Range and Fanout Maxima).
    *-----*/
    nRc = ILX_ILTIFStartNextPhase(tr, TIF_PHASE_LOADING_TARGET_RECORDS);
    if (nRc != 0)
        return nRc;

    /*-----
    * Don't skip the "Export before Import" step. Target xlator may
    * need to rebuild target app database from scratch, in which case
    * it needs to do Export before Import, even for UPD_NONE. So the
    * optimization of skipping "Export before Import" cannot be done
    * by the engine. But xlaters that don't rebuild from scratch
    * can check for ILTR_PHASE10 and UPD_NONE, and return ILTR_ERR_NORECS
    * from BEGIN to avoid unnecessary Export before Import.
    *-----*/
    // else if (ILTR_nUpdOpt == ILX_OPT_NONE)
    //     goto StartOfStepTwo;

    /*-----
    * Decide whether to enable a "shortcut" in ILXTRANS\CILTRANS. The
    * shortcut is to avoid wasting effort trying to load records from a
    * non-existent file. The flag is set here on behalf of existing
    * translators that have always been shielded in this way. Translator
    * writers should feel free to clear this flag, in their translator
    * code, if necessary.
    *-----*/
    if (prz->pTarApp->SysType == ILTB_TYPE_APP)
        switch (prz->pTarApp->AccessType)
        {
            case ILX_ACCESS_DDE:                // Dynamic Data Exchange

```

```

        case ILX_ACCESS_ODBC:                // ODBC database
        case ILX_ACCESS_NEW1:                // Unassigned
        default:
            break;

        case ILX_ACCESS_DBASE:                // dBASE database
        case ILX_ACCESS_FILE:                // File access
        case ILX_ACCESS_PDX:                 // Paradox database
        case ILX_ACCESS_CDF:                 // Ascii file type

            //----- enable shortcut in ILXTRANS\CILTRANS
            ILTR_Flags |= ILTR_FLAG_SEE_IF_FILE_EXISTS;
    }

#ifdef ILWIN
#ifndef WIN32
/*-----
 * Set both bars to 0% and label by Apps/Sections/FileNames. Note the
 * inverted source/target sense of FileNames and Access Types.
 *-----*/
    LabelProgressBars ( _ilx_globals, tr,
                        ILXGL_szTargetFile, ILXGL_szSourceFile1,
                        prz->pTarApp->AccessType, prz->pSrcApp->AccessType );
#endif
#endif

//----- Fetch Input File if this Backward Export must read
//----- data from an offboard device.
if (prz->pTarApp->SysType != ILTB_TYPE_APP)
{
    //----- enable shortcut in ILXTRANS\CILTRANS
    ILTR_Flags |= ILTR_FLAG_SEE_IF_FILE_EXISTS;

    nRc = GetRemoteInputFile ( _ilx_globals, tr,
                              prz->pTarApp,
                              ILXGL_szTargetFile,
                              ILX_PUT_TO_TARGET);

    //----- If the remote file doesn't exist, it's OK
    if (nRc == ILX_ERR_COM_NOFILE)
    {
        IL_REMOVE (ILTR_szAppFile, sFileInfo, nTempRC);
        nRc = SUCCESS;
    }

    //----- bail out if we got a nasty error
    else if (nRc != ILX_OK)
        return nRc;

    //----- Remember temp filename
    IL_STRCPY (szTempRemoteTarget, ILTR_szAppFile);
}

/*-----
 * Tell TIF to close its workfile. In ILX_V4 mode the state of the
 * TIF workfile goes through the following transitions:
 *
 * File is created in Phase 05 when doTranslate calls ILTIF[Sync]Init.
 * File is used, then closed "initially", in Phase 10 of doTranslate.
 * Each translator brackets its usage of TIF with calls to
 * ILTIFReopenFile and ILTIFCloseFileTemporarily.
 * For SmartMerge, the file is closed & deleted at the end of Phase 30,
 * when doTranslate calls ILTIFClose.
 * For Synchronization, doTranslate makes the following calls in
 * Phase 40: ILTIFReopenFile and ILTIFSyncFinishUpAndClose.
 *-----*/
nRc = ILX_ILTIFCloseFileInitially (tr);
if (nRc != 0)
    return nRc;

//----- Now run the translator to export from target.
//----- swallow 'NO DATA' condition (ILTR_ERR_NORECS)
nRc = ILX_LoadTranslator ( _ilx_globals, tr, prz->pTarApp->AppLoc);
if (nRc != 0 && nRc != ILX_ERR_NODATA)
    return nRc;

```

```

//----- Make sure that export progress bar reads 100%
#ifdef ILWIN
#ifndef WIN32
    if (ILXGL_cbProgress == NULL && ILXGL_nEnviron == ILX_ENV_NORMAL)
        SendMessage (ILTR_hFromBarWin, ILCT_MSG_UPDATE, 100, 0L);
#endif
#endif

return SUCCESS; //----- end of "Step 1" of ILX_V4 style of operation
} //----- do_Phase10_ExportFromTarget

#ifdef ILWIN
#ifndef WIN32

/*-----
 * LabelProgressBars
 * called from do_Phase10_ExportFromTarget and do_Phase20_ExportFromSource
 *-----*/
static void LabelProgressBars
( ILX_PGLOBS _ilx_globals,
  ILTR_PTRANSL tr,
  IL_PSTR szSourceFile,
  IL_PSTR szTargetFile,
  ILTB_ACC nSourceAccessType,
  ILTB_ACC nTargetAccessType )
{
    if (ILXGL_cbProgress == NULL && ILXGL_nEnviron == ILX_ENV_NORMAL)
    {
        char szText[MAX_MSG];

        //----- Reset the FROM and TO progress bars to 0%
        SendMessage (ILTR_hFromBarWin, ILCT_MSG_SETBOUNDS, 0, 0L);
        SendMessage (ILTR_hToBarWin, ILCT_MSG_SETBOUNDS, 0, 0L);

        //----- Put target section name in progress window title
        ILSTMakeString ( &hDLLInstance,
                        ILX_MSG_MERGE,
                        szText,
                        MAX_MSG,
                        ILTR_szAltSect );
        SetWindowText (ILTR_hProgWin, szText);
    }

    if (ILXGL_cbProgress == NULL)
    {
        char szSrcExt[MAX_EXT];          // Source file extension
        char szTarExt[MAX_EXT];          // Source file extension

        /*-----
        * Determine file extensions of source and target file names.
        * This information is later used to avoid showing temporary
        * file names in the progress dialog box.
        *-----*/
        szSrcExt[0] = szTarExt[0] = '\0';
        if (nSourceAccessType == ILX_ACCESS_ODBC)
            IL_splitpath (szSourceFile, NULL, NULL, NULL, szSrcExt);
        if (nTargetAccessType == ILX_ACCESS_ODBC)
            IL_splitpath (szTargetFile, NULL, NULL, NULL, szTarExt);

        //----- Place application and section name in file slot if not used
        if (szSourceFile[0] && IL_STRICMP (szSrcExt, ".tmp"))
            SetDlgItemText (ILTR_hProgWin, ILCT_PRG_FROMAPP, ILTR_szAppName);
        else
            SetDlgItemText (ILTR_hProgWin, ILCT_PRG_FROMFILE, ILTR_szAppName);

        //----- Place application and section name in file slot if not used
        if (szTargetFile[0] && IL_STRICMP (szTarExt, ".tmp"))
            SetDlgItemText (ILTR_hProgWin, ILCT_PRG_TOAPP, ILTR_szAltApp);
        else
            SetDlgItemText (ILTR_hProgWin, ILCT_PRG_TOFILE, ILTR_szAltApp);

        //----- Place source file name in progress dialog
    }
}

```

```

        if (szSourceFile[0] && IL_STRICMP (szSrcExt, ".tmp"))
            SetDlgItemText (ILTR_hProgWin, ILCT_PRG_FROMFILE, szSourceFile);

        //----- Place target file name in progress dialog
        if (ILXGL_szTargetFile[0] && IL_STRICMP (szTarExt, ".tmp"))
            SetDlgItemText (ILTR_hProgWin, ILCT_PRG_TOFILE, szTargetFile);
    }

    } //---- LabelProgressBars

#endif // WIN32
#endif // ILWIN

/*-----
 * do_Phase20_ExportFromSource (both V3 & V4) (called from doTranslate)
 *-----*/
static int do_Phase20_ExportFromSource
(
    ILX_PGLOBS _ilx_globals,
    ILX_PRECS prz,
    ILTR_PTRANSL tr,
    int EnvAttribs )
{
    int nRc;                // Return code
    int nTempRc;            // Temp return code
    IL_FILEINFO sFileInfo;  // used by IL_REMOVE

    /*-----
     * For both ILX_V3 and ILX_V4, set parameters for EXPORT FROM SOURCE
     *-----*/
    nRc = SetPhaseParams (_ilx_globals, prz, tr, EnvAttribs, ILTR_PHASE20);
    if (nRc != SUCCESS)
        return nRc;

    /*-----
     * For all ILX_V4 mode translations,
     * always cycle TIF through the 'LoadingSourceRecords' phase, whether or
     * not there are any records to load. This ensures that TIF will get a
     * chance to save any parameters associated with this phase (e.g.
     * Loading Range and Fanout Maxima).
     *-----*/
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
    {
        nRc = ILX_ILTIFStartNextPhase(tr, TIF_PHASE_LOADING_SOURCE_RECORDS);
        if (nRc != 0)
            return nRc;
    }

#ifdef ILWIN
#ifdef WIN32
    /*-----
     * Set both bars to 0% and label by Apps/Sections/FileNames.
     *-----*/
    LabelProgressBars ( _ilx_globals, tr,
                        ILXGL_szSourceFile1, ILXGL_szTargetFile,
                        prz->pSrcApp->AccessType, prz->pTarApp->AccessType );
#endif
#endif

    //----- Fetch Input File if exporting from an offboard device
    if (prz->pSrcApp->SysType != ILTB_TYPE_APP)
    {
        //---- enable shortcut in ILXTRANS\CILTRANS
        ILTR_Flags |= ILTR_FLAG_SEE_IF_FILE_EXISTS;

        nRc = GetRemoteInputFile ( _ilx_globals, tr,
                                    prz->pSrcApp,
                                    ILXGL_szSourceFile1,
                                    ILX_PUT_TO_SOURCE);

        //----- If the remote file doesn't exist, it's OK for sync
        //----- (but we still do invoke the xlator!!)
        if (nRc == ILX_ERR_COM_NOFILE &&
            ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
        {

```

```

        IL_REMOVE (ILTR_szAppFile, sFileInfo, nTempRC);
        nRc = SUCCESS;
    }

    //---- bail out if we got a nasty error
    else if (nRc != ILX_OK)
        return nRc;

    //----- Remember temp filename
    IL_STRCPY (szTempRemoteSource, ILTR_szAppFile);
}

/*-----
 * For ILX_V3 mode operations only, decide whether to load Source records
 * into TIF. For ILX_V4 mode operations this "shortcut" decision is
 * deferred to ILXTRANS\CILTRANS. We always invoke the Source
 * xlator here; this is necessary for TIF to save FanoutMaxima and maybe
 * other parameters. For ILX_V3, for Desktop Apps,
 * we don't try to load Source records if we use a file-based access
 * method for translation and the source file doesn't exist yet.
 *-----*/
if (prz->pSrcApp->SysType == ILTB_TYPE_APP)
{
    switch (prz->pSrcApp->AccessType)
    {
        case ILX_ACCESS_DDE:           // Dynamic Data Exchange
        case ILX_ACCESS_ODBC:          // ODBC database
        case ILX_ACCESS_NEW1:          // Unassigned
        default:
            break;

        case ILX_ACCESS_DBASE:         // dBASE database
        case ILX_ACCESS_FILE:          // File access
        case ILX_ACCESS_PDX:           // Paradox database
        case ILX_ACCESS_CDF:           // Ascii file type

            if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
            {
                //----- Skip the rest of Phase20 if file doesn't exist
                if (IL_DOESNT_EXIST(ILXGL_szSourceFile1))
                    return SUCCESS;
            }
            else
                //----- enable shortcut in ILXTRANS\CILTRANS
                ILTR_Flags |= ILTR_FLAG_SEE_IF_FILE_EXISTS;
    }
}

//----- Now run the translator to export from source.
nRc = ILX_LoadTranslator (_ilx_globals, tr, prz->pSrcApp->AppLoc);
if (nRc != SUCCESS)
{
    //----- swallow 'NO DATA' condition (ILTR_ERR_NORECS) for SYNC ONLY
    if (ILTR_nSynchronize && (nRc == ILX_ERR_NODATA))
        ;
    else
        //----- some other error, or not doing synchronization...
        return nRc;
}

//----- Make sure that export progress bar reads 100%
#ifdef ILWIN
    #ifndef WIN32
        if (ILXGL_cbProgress == NULL && ILXGL_nEnviron == ILX_ENV_NORMAL)
            SendMessage (ILTR_hFromBarWin, ILCT_MSG_UPDATE, 100, 0L);
    #endif // WIN32
#endif // ILWIN

return SUCCESS;
} //---- do_Phase20_ExportFromSource

/*-----
 * do_Phase30_ImportIntoTarget (both V3 & V4) (called from doTranslate)
 *-----*/

```



```

/*-----*/
static int do_Phase30_ImportIntoTarget
( ILX_PGLOBS _ilx_globals,
  ILX_PRECS prz,
  ILTR_PTRANSL tr,
  int EnvAttribs )
{
    int nRc;

    /*-----
    * For both ILX_V3 and ILX_V4, set parameters for IMPORT INTO TARGET
    *-----*/
    nRc = SetPhaseParams (_ilx_globals, prz, tr, EnvAttribs, ILTR_PHASE30);
    if (nRc != SUCCESS)
        return nRc;

    if (prz->pTarApp->SysType == ILTB_TYPE_APP)
    {
        /*--- if RemoveFile attribute is present, assert it...
        if (ILXGL_remFile || (ILTR_nAttribs & ILTB_ATT_REMFILE))
        {
            nRc = RemoveTargetFileJustOnce (_ilx_globals, tr);
            if (nRc != ILX_OK)
                return nRc;
        }
        else
        {
            /*----- Set data file name to temporary handheld file
            nRc = PreFetchTargetFile (_ilx_globals, tr, prz->pTarApp);
            if (nRc != ILX_OK)
                return nRc;
            */

            if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)    /*----- (ILX_V4)
            {
                INT16 nextPhase;

                if (ILTR_Flags & ILTR_FLAGS_SKIP_SANITIZING_STEP)
                    nextPhase = TIF_PHASE_CONFLICT_RESOLUTION;
                else
                    nextPhase = TIF_PHASE_SANITIZING_SOURCE_RECORDS;

                nRc = ILX_ILTIFStartNextPhase(tr, nextPhase);
                if (nRc != SUCCESS)
                    return nRc;
            }

            /*-----
            *
            * Culmination of Step 3 -- Invoke Target Translator.
            *
            * Target Translator is expected to do 3 things:
            *
            * 1. Sanitize Source Records
            *
            * 2. Call ILTIFEndLoad, which causes TIF to do
            *    Conflict Analysis and Resolution (CAAR) processing,
            *    which may involve use of interactive "ILCR" dialogs.
            *
            * 3. Unload TIF records that affect the Target App.
            *
            *-----*/

            nRc = ILX_LoadTranslator (_ilx_globals, tr, prz->pTarApp->AppLoc);

            return nRc;
        } //---- do_Phase30_ImportIntoTarget

        /*-----
        * do_Phase40_ImportIntoSource (V4 SYNC ONLY) (called from doTranslate)
        *-----*/
        static int do_Phase40_ImportIntoSource

```

```

    ( ILX_PGLOBS _ilx_globals,
      ILX_PRECS prz,
      ILTR_PTRANSL tr,
      int EnvAttribs )
{
    int nRc;                      // Return code
    ILX_BOOL bExists;             // Flag whether file exists

    /*-----
     * Set parameters for IMPORT INTO SOURCE
     *-----*/
    nRc = SetPhaseParams ( _ilx_globals, prz, tr, EnvAttribs, ILTR_PHASE40 );
    if ( nRc != SUCCESS )
        return nRc;

    /*-----
     * If this Backward Import wants to put data into an offboard device,
     * get a TEMP file to store the data in until we get around
     * to sending it down the wire.
     *-----*/
    if ( prz->pSrcApp->SysType != ILTB_TYPE_APP )
    {
        /*-----
         * Assign temporary file name on behalf of handheld
         * device if one not already assigned
         *-----*/
        if ( IL_STRLEN ( szTempRemoteSource ) == 0 )
            nRc = FindTempFile ( _ilx_globals,
                                ILXGL_szSourceFile1,
                                ILTR_szAppFile,
                                &bExists,
                                ILX_PUT_TO_SOURCE );

        else
        {
            IL_STRCPY ( ILTR_szAppFile, szTempRemoteSource );
            nRc = ILX_OK;
        }

        /*----- Did we succeed in getting temporary file name?
        if ( nRc != ILX_OK )
            return nRc;
        */

        nRc = ILX_ILTIFStartNextPhase( tr, TIF_PHASE_UNLOADING_TO_SOURCE );
        if ( nRc != SUCCESS )
            return nRc;

        nRc = ILX_LoadTranslator ( _ilx_globals, tr, prz->pSrcApp->AppLoc );
        return nRc;
    } //---- do_Phase40_ImportIntoSource

    /*-----
     * do_SyncFinishUp (V4 SYNC ONLY) (called from doTranslate)
     *-----*/
    static int do_SyncFinishUp
        ( ILTR_PTRANSL tr )
    {
        int nRc;

        nRc = ILX_ILTIFStartNextPhase( tr, TIF_PHASE_UNLOADING_TO_HISTORY );
        if ( nRc != SUCCESS )
            return nRc;

        /*----- Reopen the TIF file, if it was closed before translation.
        nRc = ILX_ILTIFReopenFile ( tr );
        if ( nRc != SUCCESS )
            return nRc;

        /*----- All is well, tell TIF to "wrap it up".
        nRc = ILX_ILTIFSyncFinishUpAndClose ( tr );

        return nRc;
    }

```

```

} //----- do_SyncFinishUp

/*-----
* Name:      GetRemoteInputFile
* Purpose:   Get Temporary File Name, and fetch data from
*            an offboard HandHeld device.
*-----*/
static int GetRemoteInputFile
( ILX_PGLOBALS _ilx_globals,
  ILTR_PTRANSL tr,
  ILTB_PSYSREC pSys,
  IL_PSTR szRemoteFileName,
  int nPutWhere )
{
    //----- Handheld support not implemented in WIN32 or Macintosh
    #if defined ( _WIN32 ) || defined ( ILMAC )
        return ILERROR(0, ILX_ERR_INTERNAL);
    #else

        int nRc;
        ILX_BOOL bExists;
        FARPROC pFunction;          // Pointer to DLL function

        /*-----
        * Get a temporary file name. The function returns ILX_OK if a
        * temporary file name was created. ILX_NOTOK is returned if the
        * file name already exists.
        *-----*/
        nRc = FindTempFile ( _ilx_globals,
                           szRemoteFileName,
                           ILTR_szAppFile,
                           &bExists,
                           nPutWhere );

        if (nRc != ILX_OK)
            return nRc;

        //----- Make sure the connection we have is the right kind
        if (pSys-> SysType != ILXGL_nConnectedType)
        {
            //----- Disconnect from current handheld
            nRc = ILX_DisconnectFromHandheld ( _ilx_globals );
            if (nRc)
                return nRc;

            //----- Establish connection for new type
            nRc = ILX_LoadCommDLL ( _ilx_globals, pSys-> SysType,
                                   pSys-> SysClass, pSys-> ComPort, TRUE );
            if (nRc)
                return nRc;
        }

        //----- Found a new file to retrieve from remote device
        if (!bExists)
        {
            //----- Get pointer to FILEGET function in communication DLL
            pFunction = GetProcAddress ( ILXGL_hCommDLL, "ILX_GetRemoteFile" );
            if (pFunction == NULL)
                return ILX_ERR_COMDLL;

            //----- Post progress notification if needed
            if (ILXGL_cbProgress)
                (*ILXGL_cbProgress) ( ILPSF_FILEGET_START, 0L, -1L );

            //----- Call DLL function
            nRc = (*pFunction) ( _ilx_globals, szRemoteFileName, ILTR_szAppFile );

            //----- Post progress notification if needed
            if (ILXGL_cbProgress)
                (*ILXGL_cbProgress) ( ILPSF_FILEGET_END, 0L, nRc );

            if (nRc == ILX_ERR_FILEGET || nRc == ILX_ERR_COM_NOFILE)
                CHECK_FOR_SHARP_WIZARD (pSys->SysClass, nRc);
        }
    }
}

```

```

    return nRc;

#endif // _WIN32

} //----- GetRemoteInputFile

/*-----
* Name:      PreFetchTargetFile
* Purpose:   get Temporary File Name, and optionally data, for
*            Target System that is an offboard HandHeld device.
*-----*/
static int PreFetchTargetFile
( ILX_PGLOBALS _ilx_globals,
  ILTR_PTRANSL tr,
  ILTB_PSYSREC pTarSys )
{
    //----- Handheld support not implemented in WIN32 or Macintosh
    #if defined ( _WIN32 ) || defined ( ILMAC )
        return ILERROR(0, ILX_ERR_INTERNAL);
    #else

        int nRc = ILX_OK;                // Return code
        ILX_BOOL bExists;                // Flag whether file exists
        FARPROC pFunction;                // Pointer to DLL function

        /*-----
        * Assign temporary file name on behalf of handheld
        * device if one not already set
        *-----*/
        if ( IL_STRLEN (szTempRemoteTarget) == 0 )
            nRc = FindTempFile ( _ilx_globals,
                                ILXGL_szTargetFile,
                                ILTR_szAppFile,
                                &bExists,
                                ILX_PUT_TO_TARGET );
        else
            IL_STRCPY (ILTR_szAppFile, szTempRemoteTarget);

        //----- Did we succeed in getting temporary file name?
        if (nRc != ILX_OK)
            return nRc;

        /*-----
        * Target is a Handheld.  If the required TEMP file doesn't exist,
        * decide whether we need to Pre-Fetch it from the Target Handheld.
        * Pre-fetch is necessary, for ILX_V3 mode of operation ONLY!!,
        * if we're going to do IntelliLink Reconciliation.  (For ILX_V4
        * operation the Target data has already been fetched -- that's
        * done in Phase10.  We avoid reusing the same temp file so that
        * we have a more complete trail for trouble-shooting.
        *
        * For other reconciliation methods (e.g. Sharp MERGE) there is no
        * need for us to pre-fetch the file.
        *
        * Unfortunately there is no clean way to determine whether the value
        * of ILTR_nUpdOpt conveys an IntelliLink Reconciliation Option or a
        * non-IntelliLink method.
        *
        * Two ugly options exist -- checking ILTB_ATT_SHOWNONE or checking
        * ILTB_ATT_MERGE.  Checking the 'ILTB_ATT_MERGE' system attribute is
        * little more than a surrogate for checking whether system is a SHARP
        * handheld.  With the advent of a sync-capable Sharp Zaurus Translator,
        * that check is flawed.
        *
        * Here's why it makes sense to check the SHOWNONE attribute:
        *
        * If a translator doesn't support conflict resolution, the conflict
        * resolution option is hardwired to NONE.  Since the user has no choice
        * but to accept the NONE option, we don't bother showing it to him.
        * That's the SHOWNONE==FALSE case.
        *
        * If SHOWNONE==TRUE, the NONE option is offered to the user, along
        * with one or more alternatives.  This implies that the translator

```

```

* does support conflict resolution.
*
* To maintain the effectiveness of the SHOWNONE check, we are careful
* not to give the SHOWNONE attribute to any system that does its
* own non-IntelliLink reconciliation.
*-----*/
if ( (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    && (bExists == FALSE)
    && (ILTR_nUpdOpt != UPD_NONE)
    && (pTarSys->SysAttrib & ILTB_ATT_SHOWNONE) )
{
    //----- Make sure the connection we have is the right kind
    if (pTarSys->SysType != ILXGL_nConnectedType)
    {
        //----- Disconnect from current handheld
        nRc = ILX_DisconnectFromHandheld (_ilx_globals);
        if (nRc)
            return nRc;

        //----- Establish connection for new type
        nRc = ILX_LoadCommDLL (_ilx_globals, pTarSys->SysType,
                               pTarSys->SysClass, pTarSys->ComPort, TRUE);
        if (nRc)
            return nRc;
    }

    //----- Get pointer to FILEGET function in communication DLL
    pFunction = GetProcAddress (ILXGL_hCommDLL, "ILX_GetRemoteFile");
    if (pFunction == NULL)
        return ILX_ERR_COMDLL;

    //----- Post progress notification if needed
    if (ILXGL_cbProgress)
        (*ILXGL_cbProgress) ( ILPSF_FILEGET_START, 0L, -1L);

    //----- Call DLL function
    nRc = (*pFunction) (_ilx_globals, ILXGL_szTargetFile, ILTR_szAppFile);

    //----- Post progress notification if needed
    if (ILXGL_cbProgress)
        (*ILXGL_cbProgress) ( ILPSF_FILEGET_END, 0L, nRc);

    if (nRc == ILX_ERR_FILEGET || nRc == ILX_ERR_COM_NOFILE)
        CHECK_FOR_SHARP_WIZARD (pTarSys->SysClass, nRc);

    //----- Truncate the file if any error occurred
    if (nRc != ILX_OK)
    {
        IL_HFILE hFile;
        IL_FILEINFO sFileInfo;
        int rc;

        IL_OPEN (ILTR_szAppFile, IL_ATTR_WRITE, hFile, sFileInfo, rc);
        IL_CLOSE (hFile, rc);

        //----- If file doesn't exist, we'll just create a new one
        if (nRc == ILX_ERR_FILEGET || nRc == ILX_ERR_COM_NOFILE)
            nRc = SUCCESS;
    }
}

return nRc;

#endif // #ifdef _WIN32 ... #else ...

} //----- PrefetchTargetFile

/*-----
* Name:      FindTempFile
* Purpose:   Find and remember temporary file names
* Parameters:
*   _ilx_globals - Pointer to global data
*   pOriginal - Pointer to original file name
*   pTemp - Pointer to assigned temporary file name

```

```

*   pExists - Pointer to boolean indicating if file already exists
*   nPutWhere - Says whether file is for output to Target, Source, or neither
* Returns:
*   ILX_OK - Temporary file name returned
*   ILX_NOTOK - Unable to get temporary file name
*   ILX_ERR_NOMEM - Unable to allocate dynamic memory
* Notes:
*   This function allocates temporary file names when either the
*   the source or target system is a remote device. Once a temporary
*   name is assigned, it is remembered in a file list for later
*   retrieval. If the function is called multiple times with the
*   same "original" file name, the same temporary file name assigned
*   the first time is returned rather than a new file name being
*   assigned. These temporary files contain the data that is
*   either received directly from the remote device or sent to the
*   device (if exporting to the unit).
*-----*/
static int FindTempFile ( ILX_PGLOBALS ilx_globals,
                          IL_PSTR pOriginal,
                          IL_PSTR pTemp,
                          ILX_BOOL *bExists,
                          INT16 nPutWhere )
{
    int i;                                // Loop variable
    int rc = ILX_OK;                       // Return code
    DWORD dwSize;                          // Buffer size
    ILX_PFILIST pList = NULL;              // Pointer to file list

    //----- Get pointer to file list if a list exists
    *bExists = ILX_FALSE;
    if (ILXGL_hFileList != NULL)
        pList = (ILX_PFILIST) GlobalLock (ILXGL_hFileList);

    //----- Inspect all files in list for a match on original name
    for (i = 0; i < ILXGL_nFileList; i++)
    {
        //----- Do the original file names match?
        if ( (IL_STRICMP (pList[i].szRealName, pOriginal) == 0)
            && (nPutWhere == pList[i].nPutWhere)
            && (ILXGL_bRamCard == pList[i].bRamCard) )
        {
            if (pTemp != NULL)
                IL_STRCPY (pTemp, pList[i].szTempName);
            *bExists = ILX_TRUE;
            GlobalUnlock (ILXGL_hFileList);
            return ILX_OK;
        }
    }

    //----- Release memory handle for file list
    if (ILXGL_hFileList)
        GlobalUnlock (ILXGL_hFileList);

    //----- Did not find matching file name
    ILXGL_nFileList++;
    dwSize = ILXGL_nFileList * sizeof (ILX_FILIST);

    //----- Allocate memory for new or existing file list
    if (ILXGL_hFileList == NULL || ILXGL_nFileList == 0)
        ILXGL_hFileList = GlobalAlloc (GMEM_MOVEABLE, dwSize);
    else ILXGL_hFileList = GlobalReAlloc (ILXGL_hFileList, dwSize, GMEM_MOVEABLE);
    if (ILXGL_hFileList == NULL)
        return ILX_ERR_NOMEM;

    //----- Refresh the pointer to expanded file list
    pList = (ILX_PFILIST) GlobalLock (ILXGL_hFileList);

    //----- Remember original file name or section code
    i = ILXGL_nFileList - 1;
    IL_STRCPY (pList[i].szRealName, pOriginal);
    pList[i].szTempName[0] = '\0';
    pList[i].bRamCard = ILXGL_bRamCard;
    pList[i].nPutWhere = nPutWhere;

    //----- Get a temporary file name if necessary

```

```

    if (pTemp != NULL)
    {
        if (!(ILUT_GetTempFileName (NULL, NULL, pList[i].szTempName, TRUE)))
            return ILX_NOTOK;
        IL_STRLWR (pList[i].szTempName);
        IL_STRCPY (pTemp, pList[i].szTempName);
    }

    //----- Free memory handle
    GlobalUnlock (ILXGL_hFileList);

    //----- Return without error
    return ILX_OK;
} //---- FindTempFile

/*-----
* Name:      RemoveTargetFileJustOnce
* Purpose:   Arrange for the REMFILE attribute to cause file removal ONCE.
*
* Set option to cause target file to be removed before the first import
* into it. The file is NOT removed before any subsequent imports into
* the same file. This means that the NEW callback routine will only be called
* once per data file when the REMFILE option has been set.
*-----*/
static int RemoveTargetFileJustOnce
( ILX_PGLOBALS _ilx_globals,      // Global data
  ILTR_PTRANSL tr )              // Translation structure
{
    IL_PSTR Filename = ILXGL_szTargetFile;
    ILX_BOOL bAlreadyRemovedOnce = FALSE;
    int i;                      // Loop variable
    ILX_PFILIST pList;          // Pointer to file list

    //----- Get pointer to file list if a list exists
    if (ILXGL_hFileList != IL_NULL_HANDLE)
    {
        pList = (ILX_PFILIST) GlobalLock (ILXGL_hFileList);

        //----- Inspect all files in list for a match on original name
        for (i = 0; i < ILXGL_nFileList; i++)
        {
            //----- Do the original file names match?
            if ( IL_STRINGS_EQUAL (Filename, pList[i].szRealName)
                && IL_STRING_IS_NULL(pList[i].szTempName)
                && (pList[i].nPutWhere == ILX_PUT_TO_TARGET) )
            {
                bAlreadyRemovedOnce = ILX_TRUE;
                break;
            }
        }

        //----- Release memory handle for file list
        GlobalUnlock (ILXGL_hFileList);
    }

    if (bAlreadyRemovedOnce == FALSE)
    {
        size_t ListSize;

        ILXGL_nFileList++;
        ListSize = ILXGL_nFileList * sizeof (ILX_FILIST);

        //----- Create or Expand file list
        if (ILXGL_hFileList == NULL || ILXGL_nFileList == 0)
            IL_ALLOC_MEM (ListSize, ILXGL_hFileList, pList);
        else
            IL_REALLOC_MEM (ListSize, ILXGL_hFileList, pList);

        if (pList == NULL)
            return ILX_ERR_NOMEM;

        //----- Remember original file name or section code
        i = ILXGL_nFileList - 1;
    }
}

```



```

        IL_STRCPY (pList[i].szRealName, Filename);
        IL_MAKE_STRING_NULL (pList[i].szTempName);
        pList[i].bRamCard = FALSE;
        pList[i].nPutWhere = ILX_PUT_TO_TARGET;

        //----- Free memory handle
        GlobalUnlock (ILXGL_hFileList);
    }

    if (bAlreadyRemovedOnce)
        ILTR_nAttribs &= (~ILTB_ATT_REMFILE);
    else
        ILTR_nAttribs |= ILTB_ATT_REMFILE;

    return ILX_OK;
} //----- RemoveTargetFileJustOnce

#ifdef ILWIN

/*-----
* Name:          FlHookProc
* Purpose:       Callback for message hook
* Parameters:
*   code        - Event type
*   wParam      - Unused
*   lParam      - Address of MSG structure
* Returns:       TRUE if message is processed, FALSE otherwise
*-----*/
LRESULT CALLBACK FlHookProc (int nCode, WPARAM wParam, LPARAM lParam)
{
    LRESULT lResult;
    LPMSG lpMsg = (LPMSG) lParam;

    //----- If F1 Key was pressed in a dialog box, handle it
    if ( nCode == MSGF_DIALOGBOX
        && lpMsg->message == WM_KEYDOWN
        && lpMsg->wParam == VK_F1 )
    {
        //----- Retrieve the Help Context for given window
        HWND hWin = GetParent (lpMsg->hwnd);
        int nContext = (int) GetProp (hWin, ILX_HELP_PROP);
        if (nContext != 0)
        {
            //----- Invoke context sensitive Help
            WinHelp (hWin, _szHelpFile, HELP_CONTEXT, nContext);
            return (TRUE);
        }
    }

    //----- Let someone else in the chain process this message
    lResult = CallNextHookEx (_hHook, nCode, wParam, lParam);
    return lResult;
} //----- FlHookProc

/*-----
* Name:          KillFlHookProc
* Purpose:       Destroy the message hook procedure used to trap F1
* Parameters:    None
* Returns:
*-----*/
static void KillFlHookProc ()
{
    /*-----
    * Remove the hook installed to trap F1 keys.
    *-----*/
    if (_hHook != 0)
        UnhookWindowsHookEx (_hHook);
}

/*-----

```

```

* Name:      MakeFlHookProc
* Purpose:   Install message hook used to trap F1 key
* Parameters: None
* Returns:   Sets value of global variable _hHook
*-----*/
static int MakeFlHookProc (HINSTANCE hInst)
{
    HOOKPROC lpHookProc;          // Hook procedure

    //----- Get pointer to hook procedure (Win16 SDK says to use GetProcAddress)
    lpHookProc = (HOOKPROC) GetProcAddress (hInst, "FlHookProc");
    if (lpHookProc == NULL)
        return ILERROR (11, ILX_ERR_INTERNAL);

    /*-----
    * Register hook procedure that will be used to trap F1
    * key presses in our dialogs.
    *-----*/
    #ifdef WIN32
        _hHook = SetWindowsHookEx ( WH_MSGFILTER,
                                    lpHookProc,
                                    NULL,
                                    GetCurrentThreadId () );
    #else
    {
        //----- use an entrypoint in TOOLHELP.DLL to derive
        //----- app instance handle from task info
        #ifdef MAX_PATH
            #undef MAX_PATH // suppress multiply-defined warning
        #endif
        #include <toolhelp.h>
        TASKENTRY te;          // task entry information structure
        HTASK hTask;           // current (running) task

        //----- initialize task entry structure
        _fmemset (&te, '\0', sizeof (TASKENTRY));
        te.dwSize = sizeof (TASKENTRY);

        //----- get app instance handle for current task
        hTask = GetCurrentTask ();
        if (! TaskFindHandle (&te, hTask))
            return ILERROR (12, ILX_ERR_INTERNAL);

        _hHook = SetWindowsHookEx (WH_MSGFILTER, lpHookProc, te.hInst, hTask);
    }
    #endif

    //----- complain if we can't set our hook
    if (_hHook == NULL)
        return ILERROR (13, ILX_ERR_INTERNAL);

    return SUCCESS;
} //----- MakeFlHookProc

#endif // ILWIN

```

```

#ifndef __ILTR
#define __ILTR // Signal header inclusion

/*-----
 * Name:      ILTR.H
 * Purpose: Header file for the IntelliLink Harness Library (ILTR)
 * Author: Copyright (c) IntelliLink, 1992-1995
 *-----*/

#define ILTR_CURRENT_VERSION 30

#define ILTR_VERSION_IS_AT_LEAST(ver) (ILTR_version >= ver)
#define ILTR_VERSION_IS_PRIOR_TO(ver) (ILTR_version < ver)

/*-----
 * ILTR Version Control -- for the ever-expanding "tr" structure
 * Feb 14, 1995
 *
 * It is very important to maintain as broad a range as possible of
 * cross-version compatibility, between ILX/ILWIN versions on the one
 * hand and translator (ILX*.FIL) versions on the other hand. Since the
 * "tr" structure is the primary vehicle for parameter passing between
 * ILX/ILWIN and the translators, it is the focal point for maintaining
 * compatibility.
 *
 * To maintain compatibility, we never re-arrange members inside the
 * "tr" structure, we just keep adding new ones at the end. This eats
 * into an area of reserved space, declared as "char filler[200]". But
 * as of version 30 the filler area never shrinks -- it's always a
 * 200-byte slack area after the last "active ingredient" in "tr".
 *
 * When writing translator code that references members of the "tr"
 * structure beyond "ILTR_version", the following guarantees can be
 * relied upon:
 *
 * - ILX and ILWIN always have and always will zero out the "filler" array.
 *
 * - Versions of ILX and ILWIN built before 2/10/95 used a smaller "filler"
 *   array, which extended just 34 bytes beyond "ILTR_version".
 *
 * - Versions of ILX and ILWIN built with ILTR_CURRENT_VERSION in the range
 *   1 through 19 make the "tr" structure extend just 7 bytes beyond
 *   "ILTR_cDateType". (This includes ILWIN.EXE version 3.41, since it
 *   was built with "petrified" headers dating from 12/1/95.)
 *
 * - ILX and ILWIN always have and always will set the value of
 *   ILTR_version. All ILX and ILWIN versions built prior to 2/10/95 set
 *   ILTR_version=0 implicitly by zeroing the "filler" array, which at that
 *   time included the space now occupied by ILTR_version.
 *
 * - References beyond the end of the "tr" structure, as allocated by
 *   ILX or ILWIN, will cause a GPF.
 *
 * Here is an example showing how translator code can "guard" the use a
 * BOOLEAN member of "tr" which was first invented in ILTR version 73:
 *
 *   if (ILTR_VERSION_IS_AT_LEAST(73))
 *       //---- ILTR_futureBool is defined and can be referenced
 *   else
 *       //---- ILTR_futureBool is not defined. Reference may cause GPF.
 *       //---- zero value does NOT mean FALSE!!
 *
 * Please follow the convention of using the ILTR_VERSION_IS_AT_LEAST()
 * macro for such checking, with no embedded spaces in the macro
 * invocation, so that the following type of search will work correctly:
 *
 *   grep ILTR_VERSION_IS_AT_LEAST(73) *.c*
 *-----*/

//----- Special handling for C++ code
#ifdef __cplusplus
    extern "C" {
#endif // __cplusplus

```

```

/*-----
 * Header files.
 *-----*/

//----- Windows headers
#ifdef ILWIN
    #if !defined( _WINDOWS)
        #include <windows.h>
        #define _WINDOWS
    #endif // _WINDOWS
    #include "ilctl.h"
#endif // ILWIN

//----- IntelliLink headers
#include "iltrerr.h"
#include "iltime.h"
#include "ilutil.h"
#include "ilst.h"

/*-----
 * Next definition tells ILIF not to use a Global Variable for it's
 * "play area". Instead the "play area" is dynamically allocated and
 * freed in EXPORT.C and IMPORT.C, to avoid reentrancy problems.
 *-----*/
#define ILIF_GLOBALS_PTR ILTR_pILIF_Globals

#include "ilif.h"
#include "iliferr.h"
#include "iltb.h"
#include "ilrpt.h"
#include "ilcr.h"

//----- Declare pointer to Translation structure
typedef struct _transl IL_DIST *ILTR_PTRANSL;

#include "iltif.h"

/*-----
 * Symbolic constants.
 *-----*/

//----- Size limits
#define ILTR_MAX_FIELDLNGTH 32766 // Max STRLEN for a field value
#define ILTR_MAX_AREACODE 32 // Size of area code
#define ILTR_MAX_COUNTRY 4 // Size of country code
#define ILTR_MAX_DEFPHONE 2 // Size of default indicator
#define ILTR_MAX_PHONEXT 17 // Size of phone extension
#define ILTR_MAX_FLAGS 10 // Number of field flags
#define ILTR_MAX_FLDERR 10 // Max number of log errors
#define ILTR_MAX_ITEMS 25 // Max field items
#define ILTR_MAX_PHONE 101 // Size of phone number
#define ILTR_MAX_PREFIX 17 // Size of field prefix
#define ILTR_MAX_PSWD 25 // Size of password
#define ILTR_MAX_TIMER_RECS 10 // Default records per timer
#define ILTR_MAX_SECT MAX_APP_NAME // Size of a section
#define ILTR_MAX_TERM 3 // Size of line terminator
#define ILTR_MAX_TYPEDESC 21 // Size of type description
#define ILTR_MAX_SST_COUNT 20 // Max # of subtypes for one
// section type for One App
#define ILTR_MAX_TAG_LEN 19 // max STRLEN of tag

//----- Filter-specific limits
#define ILTR_MAX_VALUE 256 // Size of filter item value
#define ILTR_MAX_COND 28 // Size of filter item condition
#define ILTR_MAX_CONDNUM 41 // Num of filter item conditions

/*-----
 * System-defined fields: these are fields that are used internally
 * but don't appear in the Field Lists that come from TABLES.ITB or
 * 'WhatFields'.
 *-----*/

//----- System-defined HIDDEN fields defined in ILRPT.H
#ifdef __ILRPT // Only include header once
    #define ILTR_MAX_FLDNAME 31 // Max size of field names

```

```

#define ILTR_REP_BASIC      "_repBasic"      // Basic repeat field
#define ILTR_REP_XDATE      "_repExcl"      // Exclusion date field
#define ILTR_APP_DATA      "_appData"      // Application binary field
#endif

//----- another System-defined HIDDEN field // this field conveys a record's
#define ILTR_SUB_TYPE      "_subType"      // original section subType

//----- Field-list-defined HIDDEN fields for Sync. Capable translators
#define ILTR_FLD_RECORD_ID  "_RecordID"     // Record ID hidden field
#define ILTR_FLD_UNIQUE_ID  "_UniqueID"     // Unique ID hidden field
#define ILTR_FLD_DELTA      "_Delta"        // for "Fast Sync" Add/Chg/Del

//----- counts of System-defined fields
#define ILTR_EXTRA_FIELDS_ALWAYS 2          // for all sections
#define ILTR_EXTRA_FIELDS_FOR_REPEAT 2      // for Appts & Todos

//----- Values for use in the "_Delta" field, for Fast Sync,
//----- to answer the question "What has happened since the last sync?"
#define ILTR_DELTA_ADD      "A"             // this record has been ADDED
#define ILTR_DELTA_CHANGE   "C"             // this record has been CHANGED
#define ILTR_DELTA_DELETE   "D"             // this record has been DELETED

//----- Character values used in "_Delta" field, for Fast Sync,
#define ILTR_CDELTA_ADD     'A'             // this record has been ADDED
#define ILTR_CDELTA_CHANGE  'C'             // this record has been CHANGED
#define ILTR_CDELTA_DELETE  'D'             // this record has been DELETED

//----- String constants
#define ILTR_RES_FILE       "ILX.STR"       // Resource file name
#define ILTR_DEFAULT_TERM   "\r\n"         // Default line terminator
#define ILTR_ENV_PROP       "ILTR_ENV_PROP" // Property name of ILTR_ENV value

//----- "Data" directory name -- sub-directory under the IntelliLink
//----- installation directory where we keep persistent internal DATA
//----- files (NOT settings files or LOG files or TMP files or USER data)
#define ILTR_DATA_DIR       "ILDATA"        // IntelliLink Data Directory

//----- Command types - used only in DOS command file
#define ILTR_CMD_APPFILE    "APPFILE"       // Application file
#define ILTR_CMD_CDFNAMES   "CDFNAMES"     // CDF names in first record
#define ILTR_CMD_CDFSEP     "CDFSEP"       // CDF field separator
#define ILTR_CMD_COMMAND    "COMMAND"      // Command file
#define ILTR_CMD_DIRECTION  "DIRECTION"    // Translation direction
#define ILTR_CMD_EXISTS     "EXISTS"       // File exists flag
#define ILTR_CMD_FUNCTION   "FUNCTION"     // Function type
#define ILTR_CMD_LOG        "LOG"          // Log file flag
#define ILTR_CMD_LOGFILE    "LOGFILE"      // Log file
#define ILTR_CMD_RANGE      "RANGE"        // Future or all
#define ILTR_CMD_RECONCILE   "RECONCILE"    // Reconciliation option
#define ILTR_CMD_SOURCE     "SRCAPP"       // Source application ID
#define ILTR_CMD_TARGET     "TARAPP"       // Target application ID
#define ILTR_CMD_TYPE       "TYPE"         // File type
#define ILTR_CMD_WORKFILE    "WORKFILE"    // Intermediate file

//----- Control character constants
#define ILTR_BSLASH_CHAR    '\\\ '        // BACKSLASH
#define ILTR_BSLASH_STR     "\\\"         // BACKSLASH string
#define ILTR_COLON_CHAR     ':'            // COLON
#define ILTR_COLON_STR      ":"            // COLON string
#define ILTR_CR_CHAR        0xd           // RETURN
#define ILTR_CR_STR         "\r"          // RETURN string
#define ILTR_CRLF_STR       "\r\n"        // CRLF pair
#define ILTR_CTRLA_CHAR     '\a'          // CTRL-A
#define ILTR_CTRLB_CHAR     '\b'          // CTRL-B
#define ILTR_CTRLZ_CHAR     '\z'          // CTRL-Z
#define ILTR_EOS_CHAR       '\xFF'        // Internal END-OF-LINE char
#define ILTR_EOS_STR        "\xFF"        // same thing as a string
#define ILTR_LF_CHAR        0xa           // NEWLINE
#define ILTR_NULL_CHAR      '\0'          // NULL
#define ILTR_NULL_STR       "\0"          // NULL string
#define ILTR_PERIOD_CHAR    '.'            // PERIOD
#define ILTR_PERIOD_STR     "."            // PERIOD string
#define ILTR_SPACE_CHAR     ' '           // SPACE
#define ILTR_SPACE_STR      " "           // SPACE string

```

```

#define ILTR_TAB_CHAR          0x9          // TAB
#define ILTR_TAB_STR          "\t"         // TAB string
#define QUOTE_OFF             0x0000      // Quote off indicator
#define QUOTE_ON              0xFFFF      // Quote on indicator

//----- Processing stages
#define ILTR_BEGIN            600          // Begin stage
#define ILTR_WHILE            601          // While loop
#define ILTR_END              602          // End stage
#define ILTR_QUIT             603          // Quit stage
#define ILTR_DONE             604          // Done stage

/*-----
 * Phases of SmartMerge and Synchronization Jobs.
 * Nonzero values are put into "ILTR_phase" for ILX_V4 operation only.
 * But other values are used locally in ilx_v3\xlite.c for both V3 and V4.
 *-----*/
#define ILTR_PHASE_ILX_V3_MODE 00          // Selects ILX_V3 (ILIF-based) operation
#define ILTR_PHASE01           01          // loading field lists (V3+V4)
#define ILTR_PHASE05           05          // getting field defs from SOURCE (V4)
#define ILTR_PHASE10           10          // exporting from TARGET (V4)
#define ILTR_PHASE20           20          // exporting from SOURCE (V3+V4)
#define ILTR_PHASE30           30          // importing into TARGET (V3+V4)
#define ILTR_PHASE40           40          // importing into SOURCE (sync only)(V4)

//----- Windows message IDs
#define ILTR_BEGIN_MSG        WM_USER+ILTR_BEGIN // Begin message type
#define ILTR_WHILE_MSG        WM_USER+ILTR_WHILE // While message type
#define ILTR_END_MSG          WM_USER+ILTR_END   // End message type
#define ILTR_QUIT_MSG         WM_USER+ILTR_QUIT  // Quit message type
#define ILTR_DONE_MSG         WM_USER+ILTR_DONE  // Done message type

//----- Miscellaneous constants
#define ILTR_NOT_USED         0              // Field not used

/*-----
 * MapField field index values for unmapped fields
 *-----*/
#define ILTR_UNMAPPED         (-1)           // Unmapped field
#define ILTR_UNMAPPED_BUT_TAGGED 0x4000      // Unmapped TAGGED field

//----- Return status from Import and Export
#define ILTR_SKIP_WRITE       0x0001        // Do not write out record
#define ILTR_SKIP_METER       0x0002        // Do not update progress meter
#define ILTR_SKIP_LOG         0x0004        // Do not update log file
#define ILTR_SKIP_SHOW        0x0006        // Do not update meter or log
#define ILTR_SKIP_BOTH        0x0007        // Skip record and output
#define ILTR_SKIP_ALL         0x0007        // Same as ILTR_SKIP_BOTH

//----- Translation attributes.
#define ILTR_ATT_REMFILE      0x0001        // Remove file before import

//----- Filter defines
#define IL_FILTER_ILLEGAL     -1             // Filter is illegal
#define IL_FILTER_BLANK       -2             // Filter is blank
#define IL_FILTER_OK          -3             // Filter is OK
#define IL_FILTER_NOMEM       -4             // No mem to alloc filter struct
#define IL_FILTER_NOOPEN      -5             // Couldn't open filter file

//----- Random Flag bits passed in ILTR_Flags:

/*-----
 * The KEEPFILERS flag causes us NOT to clean up temporary files
 * at the end of a job. This includes intermediate files (ILIF & TIF)
 * as well as files transferred to or from handheld units.
 *-----*/
#define ILTR_FLAG_KEEFILES    0x00000001L

/*-----
 * The SKIP_SANITIZING_STEP flag, applicable only to ILX_V4 jobs,
 * says that Source Data Records emitted by the Source Translator
 * will NOT be "sanitized" by the Target Translator before Conflict
 * Resolution is performed. This flag affects the behavior of TIF,
 * during Source Export (PHASE20), and affects the behavior of
 * IMPORT.C, during Target Import (PHASE30).
 *-----*/

```

```

/*-----*/
#define ILTR_FLAGS_SKIP_SANITIZING_STEP 0x00000002L

/*-----
 * The ILTR_NO_SYNTHETIC_MATCHES flag is used to turn off the SYNC
 * feature that matches up Recurring Master items with Fanned Instances.
 *-----*/
#define ILTR_NO_SYNTHETIC_MATCHES 0x00000004L

/*-----
 * The ILTR_FLAG_FANNING flag is set and cleared by ILRepeatItem
 * at the start and conclusion of a fanning operation.
 *-----*/
#define ILTR_FLAG_FANNING 0x00000008L

/*-----
 * The ILTR_FIELD_LEVEL_LOGGING flag is used to turn on very verbose
 * logging of field values as they are processed by ILXTRANS.
 *-----*/
#define ILTR_FIELD_LEVEL_LOGGING 0x00000010L // 16.

/*-----
 * The ILTR_DISABLE_SST_TAGGING flag tells the harness not to attach
 * Section SubType tags to user-visible fields in application databases.
 *-----*/
#define ILTR_DISABLE_SST_TAGGING 0x00000020L // 32.

/*-----
 * The ILTR_DISABLE_SST_FILTERING flag tells the harness not to
 * filter out any records due to Section SubType mismatches.
 *-----*/
#define ILTR_DISABLE_SST_FILTERING 0x00000040L // 64.

/*-----
 * The ILTR_DISABLE_SST_IF_UNMAPPED flag tells the harness to turn off
 * SST Tagging & Filtering whenever a TAGGED field is UNMAPPED. When
 * this flag is NOT used, special treatment is accorded to UNMAPPED
 * TAGGED fields, so that most of the world is duped into thinking that
 * such fields ARE MAPPED. When this flag IS used, all such special
 * treatment is shut off entirely, and all of SST is then at the mercy
 * of field mapping. Recommendation: leave this flag OFF.
 *-----*/
#define ILTR_DISABLE_SST_IF_UNMAPPED 0x00000080L // 128.

/*-----
 * The ILTR_FLAG_MACINTOSH flag says we're running on a Macintosh!
 *-----*/
#define ILTR_FLAG_MACINTOSH 0x00000100L // 256.

/*-----
 * The ILTR_FLAG_MIXED_WIN3216 flag says that a WIN32 engine is running
 * a translation where one or both of the translators are 16-bit.
 *-----*/
#define ILTR_FLAG_MIXED_WIN3216 0x00000200L // 512.

/*-----
 * The ILTR_FLAG_APPEND_TO_LOGS flag tells components such as TIF that
 * they should keep appending to existing logfiles rather than
 * overwriting them.
 *-----*/
#define ILTR_FLAG_APPEND_TO_LOGS 0x00000400L // 1024.

/*-----
 * The ILTR_FLAG_SOURCE_DEL_OUTRANGE flag, for Synchronization, tells
 * TIF that the user wants all records that fall outside the Date Range
 * deleted from the Source Application.
 *-----*/
#define ILTR_FLAG_SOURCE_DEL_OUTRANGE 0x00000800L // 2048.

/*-----
 * The ILTR_FLAG_TARGET_DEL_OUTRANGE flag, for Synchronization, tells
 * TIF that the user wants all records that fall outside the Date Range
 * deleted from the Target Application.
 *-----*/
#define ILTR_FLAG_TARGET_DEL_OUTRANGE 0x00001000L // 4096.

```



```

/*-----
 * The ILTR_FLAG_UNATTENDED_MODE flag is derived from the ILX flag
 * ILX_FLAG_UNATTENDED_MODE. These flags are to be set when the user
 * is not physically present (may be dialing in remotely) and cannot
 * respond to ANY dialogs or message boxes. First used by ILPILOT.
 *-----*/
#define ILTR_FLAG_UNATTENDED_MODE      0x00002000L    // 8192.

/*-----
 * The ILTR_FLAG_ENGINE_TRACE flag is used to turn on tracing of
 * engine operations. Trace output is written to ILERRORS.LOG.
 *-----*/
#define ILTR_FLAG_ENGINE_TRACE          0x00004000L    // 16384.

/*-----
 * The ILTR_FLAG_FIRST_XLATE flag, set by xlate.c, is TRUE for the first
 * source-to-target translation in a session, and FALSE for any
 * subsequent translations done in the same session.
 *-----*/
#define ILTR_FLAG_FIRST_XLATE           0x00008000L    // 32768.

/*-----
 * Set this flag to cause an IMPORT operation to call the Chooser.
 *-----*/
#define ILTR_FLAG_IMPORT_SELECTED        0x00010000L    // 65536.

/*-----
 * The ILTR_FLAG_SEE_IF_FILE_EXISTS flag, set by xlate.c, tells code
 * in ILXTRANS\CILTRANS to do an existence check on ILTR_szAppFile,
 * and to avoid calling the datastore "Open" function if the file
 * does not exist. This flag is applicable to EXPORT phases only!!
 *-----*/
#define ILTR_FLAG_SEE_IF_FILE_EXISTS     0x00020000L    // 131072.

/*-----
 * The ILTR_FLAG_REMOTE flag is set when we're operating under a remote
 * server, serving a client (e.g. under Transit97).
 *-----*/
#define ILTR_FLAG_REMOTE                 0x00040000L    // 262144.

/*-----
 * The Quit Flag bit, in ILTR_uDlgProgFlags, is used under Windows to
 * make sure that we break out of the translation Message Loop when
 * the time comes, even if the ILTR_QUIT_MSG is dispatched by someone
 * else's Message Loop (and therefore isn't seen by our Message Loop.)
 *-----*/
#define ILTR_QUIT_FLAG                   0x0010
#define ILTR_QUIT_FLAG_IS_SET (ILTR_uDlgProgFlags & ILTR_QUIT_FLAG)
#define SET_ILTR_QUIT_FLAG ILTR_uDlgProgFlags |= ILTR_QUIT_FLAG
#define CLEAR_ILTR_QUIT_FLAG ILTR_uDlgProgFlags &= ~ILTR_QUIT_FLAG

/*-----
 * The following 'PRIO_TYPE' values are used in the ILTR_TodoPriorityType
 * member of the "tr" structure to control handling of ToDo Priorities by
 * TIF and by translators.
 *
 * Under type "NUMERIC_0" translators exchange numeric priorities that
 * are normalized into the range 0-99. In many cases normalization is done
 * by subtracting 1 from the native value.
 *
 * Under type "NUMERIC_1" translators exchange un-normalized
 * numeric priorities.
 *
 * Under type "PASS_THRU" all priority values are passed through unmodified.
 *
 * Type "SHARP_SPLUS" allows Sharp Wizards and Schedule+ 7.0 to exchange
 * values 1-36, representing S+ display values 1-9,A-Z, and 29.
 *-----*/
#define ILTR_PRIO_TYPE_NUMERIC_0 0
#define ILTR_PRIO_TYPE_NUMERIC_1 1
#define ILTR_PRIO_TYPE_PASS_THRU 2
#define ILTR_PRIO_TYPE_SHARP_SPLUS 3

//----- Date defines

```

```

#define MAX_DATE_SPEC      11          // Max size of date specification
#define MAX_SEP            3           // Max size of date separator field

//----- "Must match" flags for ILComparePhone
#define ILTR_MATCH_COUNTRY 0x80       // Country codes must match
#define ILTR_MATCH_AREA    0x04       // Area codes must match
#define ILTR_MATCH_PHONE   0x02       // Phone number must match
#define ILTR_MATCH_EXT     0x01       // Phone extension must match

/*-----
 * Types.
 *-----*/

typedef INT16 ILTR_ENV;               // IntelliLink environments
#define ILTR_ENV_ILWIN      0         // IntelliLink for Windows
#define ILTR_ENV_LITE       1         // IntelliLink Lite
#define ILTR_ENV_MAGICXCHANGE 2       // MagicXChange
#define ILTR_ENV_WINPAD     3         // Microsoft WinPad
#define ILTR_ENV_ACHATES    4         // Intel Achates

typedef INT16 ILTR_ACTION;            // Action types
#define ILTR_ACT_ADD        0         // Add an item
#define ILTR_ACT_FAN        1         // Fan an item
#define ILTR_ACT_IGNORE     2         // Ignore an item
#define ILTR_ACT_READ       3         // Read an item
#define ILTR_ACT_REPLACE    4         // Replace an item
#define ILTR_ACT_SKIP       5         // Skip an item
#define ILTR_ACT_UPDATE     6         // Update modified fields
#define ILTR_ACT_FILTER     7         // Filter an item
#define ILTR_ACT_RANGE      8         // Fail range check
#define ILTR_ACT_DELETE     9         // Delete an item (synchro)

#define ILTR_ACT_LOADED INTO_TIF 19    // when *cbPut() writes to TIF

//----- Direction of Merge
typedef INT16 ILTR_DIRECTION;         // Direction type
#define ILTR_IMPORT         0         // Import operation
#define ILTR_EXPORT         1         // Export operation

//----- Types for password dialog
typedef INT16 ILTR_PASSWORD_TYPE;     // Password dialog type
#define ILTR_PSWD_FILE_PSWD 0         // File password
#define ILTR_PSWD_USER_NAME 1         // User name
#define ILTR_PSWD_USER_PSWD 2         // User password

//----- Function or category types (defined in terms of ILTB types)
typedef ILTB_SEC ILTR_FUNCTION;        // Category type
#define ILTR_APPT           ILTB_SEC_APPT // Appointments
#define ILTR_DATA           ILTB_SEC_DB   // Database
#define ILTR_MEMO           ILTB_SEC_MEMO // Memos
#define ILTR_PHONE          ILTB_SEC_PHONE // Phone book entries
#define ILTR_TODO           ILTB_SEC_TODO // Todo items
#define ILTR_GROUPS         ILTB_SEC_GROUPS // Phone Groups or Lists
#define ILTR_OUTLINE        ILTB_SEC_OUTLINE // Used for ECCO outlines
#define ILTR_CALL           ILTB_SEC_CALL // Calls
#define ILTR_SPREAD         ILTB_SEC_SPREAD // Spreadsheet
#define ILTR_EXPENSE        ILTB_SEC_EXPENSE // Expense

//----- Appointment Range
typedef ILX_RANGE ILTR_RANGE;          // Appointment range
#define ILTR_RANGE_ALL      ILX_RANGE_ALL // All
#define ILTR_RANGE_FUTURE   ILX_RANGE_FUTURE // Future
#define ILTR_RANGE_NONE     ILX_RANGE_NONE // None

/*-----
 * These values are provided only for backward compatibility for
 * older translators (before table-driven stuff was invented).
 * The values must correspond with the values in ILTB_REC, which
 * is defined in iltbl.h. Do not add new values here or change
 * them (unless of course corresponding changes are being made
 * to ILTB_REC for some inane reason). These should be obsoleted
 * someday.
 *-----*/
typedef ILX_OPTION ILTR_UPDOPT;        // Update option settings
#define UPD_REPLACE         ILX_OPT_REPLACE // Replace existing item

```

```

#define UPD_IGNORE      ILX_OPT_IGNORE    // Ignore incoming item
#define UPD_NOTIFY      ILX_OPT_NOTIFY    // Notify user of conflict
#define UPD_INSERT      ILX_OPT_INSERT    // Insert duplicate item
#define UPD_UPDATE      ILX_OPT_UPDATE    // Update selected fields
#define UPD_MERGE       ILX_OPT_MERGE     // Merge new items (Sharp)
#define UPD_CANCEL      ILX_OPT_ACCEPT_1  // Cancel current translation
#define UPD_VERIFY      ILX_OPT_ACCEPT_2  // Recheck for conflicts
#define UPD_NONE        ILX_OPT_NONE     // No reconciliation
#define UPD_DELETE      ILX_OPT_DELETE    // Delete record
#define UPD_DELTA_ACK   ILX_OPT_DELTA_ACK // a FastSync Unload Action Code

/*-----
 * Enumerated types.
 *-----*/

typedef enum                // Match return values
{
    ILTR_MATCH_FAIL        = 1,          // Internal failure
    ILTR_MATCH_NONE        = 2,          // No match found
    ILTR_MATCH_FOUND       = 3,          // Match found
    ILTR_MATCH_CONFLICT    = 4,          // Conflict found
} ILTR_MATCH;

typedef enum                // Notification options
{
    ILTR_ACCEPT,            // Accept new data
    ILTR_ADD,              // Add new data
    ILTR_DISCARD,          // Discard new data
    ILTR_EXIT,             // Abort reconciliation
    ILTR_OVERWRITE,        // Replace existing data
    ILTR_REVERTALL,        // Revert to original fields
    ILTR_SAVEALL,          // Save all new fields
    ILTR_CONTINUE,         // View next conflict
} ILTR_NOTIFY_CHOICE;

typedef enum                // Sides to strip blanks from
{
    ILTR_LEADING    = 1,          // Strip leading blanks
    ILTR_TRAILING   = 2,          // Strip trailing blanks
    ILTR_BOTH       = 3,          // Strip blanks from both sides
} ILTR_SIDES;

typedef enum                // Field sort order
{
    ILTR_SORT_BY_LABEL,          // Sort fields in label order
    ILTR_SORT_BY_NAME           // Sort fields in name order
} ILTR_SORT_ORDER;

/*-----
 * Data type definitions.
 *-----*/

//----- Field name structure
typedef struct
{
    BOOL16 Mapped;                // Is field mapped?
    char Text[ILTR_MAX_FLDNAME]; // Field name
    INT16 ItemNo;                // Item Number of field
} ILTR_LSTITEM, IL_DIST *ILTR_PLSTITEM;

//----- Field list structure
typedef struct
{
    INT16 Count;                  // Number of fields in list
    IL_HANDLE handle;            // Handle to field list
    ILX32_16PAD(pad_01,2)        // Padding for tr purposes
    ILTR_PLSTITEM Name;          // List of field names
} ILTR_FLDLST, IL_DIST *ILTR_LSTPTR;

typedef INT16 ILTR_NDX;          // Field node index

//----- Entry in field map
typedef struct
{
    //----- Field attributes

```

```

    char IntName[ILTR_MAX_FLDNAME];           // Internal field name
    char ExtName[ILTR_MAX_FLDNAME];           // External field name
    INT16 ItemNo;                             // Item number within field
    char Type;                                // Field type
    long Width;                               // Field width
    INT16 Term;                               // Field terminator
    char Label[ILTR_MAX_PREFIX];              // Optional field prefix
    char TypeDesc[ILTR_MAX_TYPEDESC];         // Optional type description
    INT16 Assoc;                              // Index of associated field
    unsigned long Attribs;                    // Field attributes

    //----- Index numbers used for list box operations
    ILTR_NDX Index;                          // Index number of node

    //----- Pointers to fields
    ILTR_NDX NextField;                      // Pointer to next field
    ILTR_NDX PriorField;                    // Pointer to prior field
    ILTR_NDX MapField;                      // Pointer to mapped field
} ILTR_FIELD, IL_DIST *ILTR_FLDPTR;

//----- Field map table
typedef struct
{
    INT16 nSource;                          // Count of source fields
    INT16 nTarget;                          // Count of target fields
    INT16 nMapStatus;                       // Map status indicator
    char sName[ILTR_MAX_FLDNAME];           // Template name
    ILTR_NDX ApptDate;                      // Index of appt date field
    ILTR_NDX ShowSource;                    // Index of source "show" field
    ILTR_NDX ShowTarget;                    // Index of target "show" field
    ILTR_NDX TopSource;                     // Index of top source field
    ILTR_NDX TopTarget;                     // Index of top target field
    IL_HANDLE hSource;                      // Handle to source fields
    ILX32_16PAD(pad_01,2)                   // Padding for tr purposes
    IL_HANDLE hTarget;                      // Handle to target fields
    ILX32_16PAD(pad_02,2)                   // Padding for tr purposes
    ILTR_FLDPTR pSource;                    // Pointer to top source field
    ILTR_FLDPTR pTarget;                    // Pointer to top target field
} ILTR_FLDMAP, IL_DIST *ILTR_PFLDMAP;

//----- Intermediate record structure
typedef struct
{
    UINT16 width;                           // Allocated size
    IL_HANDLE handle;                       // Handle to record
    ILX32_16PAD(pad_01,2)                   // Padding for tr purposes
    IL_PSTR buffer;                         // Pointer to buffer
} ILTR_BUFFER, IL_DIST *ILTR_PBUFFER;

//----- SST (Section SubType) LIST struct
typedef struct
{
    INT16 sstCount;
    BYTE sstList[ILTR_MAX_SST_COUNT];
}
ILTR_SSTLIST, IL_DIST *ILTR_PSSTLIST;

//----- Field list information structure used ONLY within module LOADFLDS.C
typedef struct
{
    ILTB_ID nOriginalSourceID;              // Original source application ID
    ILTB_ID nOriginalTargetID;              // Original target application ID
    INT16 nExtraFields;                     // Number of fields added to field lists
    ILTR_FLDMAP sFieldMap;                  // Field map table
    ILTR_BUFFER sSourceExportCharMap;       // Source Export CharMap buffer
    ILTR_BUFFER sSourceImportCharMap;       // Source Import CharMap buffer
    ILTR_SSTLIST sOriginalSourceSSTList;    // O. Source List of Section SubTypes
    ILTR_SSTLIST sOriginalTargetSSTList;    // O. Target List of Section SubTypes
    ILTR_BUFFER sTargetExportCharMap;       // Target Export CharMap buffer
    ILTR_BUFFER sTargetImportCharMap;       // Target Import CharMap buffer
} ILTR_TABLEINFO, IL_DIST *ILTR_PTABLEINFO;

//----- Structure used to hold field errors
typedef struct
{

```

```

    INT16 nError;                // Error code
    char szField[ILTR_MAX_FLDNAME]; // Field name
} ILTR_FLDERR, IL_DIST *ILTR_PFLDERR;

//----- Structure passed via pointer to translator ILWhatFieldsEx function
typedef struct                    // ILWhatFields parameter block
{
    IL_PSTR    pszAppFile;        // Pointer to data file name
    ILTB_ID    nCharMapID;        // ID of character map
    ILTB_PHNDL phSysTable;        // Pointer to system table handle
    IL_LPHANDLE phFldList;        // Pointer to field list handle
    ILPINT     pnFldCount;        // Pointer to field count
    char        cDelimiter;       // ASCII delimiter character
    IL_PANY     pXtraData;        // Pointer to translator XtraData
} ILX_WFParams, IL_DIST *ILX_PWFParams;

//----- Structure passed via pointer to ILGetPassword function
typedef struct                    // ILGetPassword parameter block
{
    IL_PSTR    pszAppFile;        // Ptr to application file name
    IL_PSTR    pszSectName;       // Ptr to application section name
    IL_PSTR    pszPassword;       // Ptr to caller's password buffer
} ILX_PswdParams, IL_DIST *ILX_PPswdParams;

//----- Declare pointers to Repeat structure
typedef struct _repeat IL_DIST *ILTR_PREPEAT;

//----- Function types
typedef int (IL_DECL IL_DIST *ILTR_PIO) // Pointer to Import or Export
( ILTR_PTRANSL );                       // Pointer to translation record
typedef int (IL_DECL IL_DIST *ILTR_CBFUN) // Pointer to callback
( ILTR_PTRANSL,                         // Pointer to translation record
  void IL_DIST * IL_DIST * );          // Pointer to application data
typedef int (IL_DECL IL_DIST *ILTR_CBREC) // Pointer to reconcile callback
( ILTR_PTRANSL,                         // Pointer to translation record
  IL_PSTR,                             // Field name to reconcile
  IL_PSTR,                             // Field value to reconcile
  void IL_DIST *,                      // Unique record identifier
  void IL_DIST * IL_DIST * );          // Pointer to application data
typedef int (IL_DECL IL_DIST *ILTR_CBREP) // Pointer to repeat callback
( ILTR_PTRANSL,                         // Pointer to translation record
  void IL_DIST * IL_DIST *,            // Pointer to application data
  ILTR_PREPEAT );                     // Pointer to repeat structure

//----- Filter item structure and pointer
typedef struct
{
    char name[ILTB_MAX_NAME];        // Filter item data
    char label[ILTR_MAX_FLDNAME];    // User visible name
    INT16 itemNo;                    // Internal field label
    ILX_TYPE type;                   // Item number within field
    INT16 cond;                      // Field type
    char value1[ILTR_MAX_VALUE];     // Condition for filter item
    char value2[ILTR_MAX_VALUE];     // First value for filter item
    long ivaluel;                    // Second value for filter item
    long ivalue2;                    // First value for filter item
    INT16 pass;                      // Second value for filter item
    // Did this test pass
} ILTR_FILTER_ITEM, IL_DIST *ILTR_PFILTER_ITEM;

//----- Filter structure and pointer
typedef struct
{
    ILTB_ID secType;                 // Filter data
    ILTB_ID mapID;                   // Section type
    ILTB_ID sysID;                   // Field map ID
    char name[ILTB_MAX_NAME];        // System ID
    char appName[ILTB_MAX_NAME];     // Name of the filter
    char secName[ILTB_MAX_NAME];     // Application name
    INT16 itemNum;                   // Section name
    BOOL16 allcond;                  // Number of items for this filter
    BOOL16 bReversed;                // Do all items have to pass
    ILTR_FILTER_ITEM items[];        // Are field lists reversed?
    // Variable array of items
} ILTR_FILTER, IL_DIST *ILTR_PFILTER;

//----- Structures used to prompt user for section name

```

```

typedef struct
{
    char szName[ILTR_MAX_SECT];           // Section name
    long lKey;                             // Item Data
} ILTR_SECTIONS;                          // Sections info

typedef struct
{
    IL_HINST hInst;                       // Instance handle
    IL_HWIN hWin;                         // Window handle
    char szCurWD[ILTB_MAX_PATH];         // Working directory
    char szHelpFile[ILTB_MAX_PATH];      // Helpfile Pathname
    char szFile[ILTB_MAX_PATH];          // File name
    char szSect[ILTR_MAX_SECT];          // Section value
    long lKey;                             // Item Data
} ILTR_SECT, IL_DIST *ILTR_PSECT;

typedef struct
{
    INT16 nEntries;                       // Number of section names
    ILTR_SECTIONS lpItems[];              // Array of section names
} ILTR_SECTS, IL_DIST *ILTR_PSECTS;

//----- Structure used to prompt user for password
typedef struct
{
    IL_HINST hInst;                       // Instance handle
    IL_HWIN hWin;                         // Window handle
    char szCurWD[ILTB_MAX_PATH];         // Working directory
    char szHelpFile[ILTB_MAX_PATH];      // Helpfile Pathname
    char szFile[ILTB_MAX_PATH];          // File name
    char szPswd[ILTR_MAX_PSWD];          // Password value
    ILTR_PASSWORD_TYPE nPassword;        // File passwd, user name/passwd?
} ILTR_PSWD, IL_DIST *ILTR_PPSWD;

//----- Translator driver (ILImport and ILExpert) function type.
typedef int (IL_DECL *ILTR_PDRIVER)      // Pointer to Import or Export
( ILTR_PTRANSL );                       // Pointer to translation data

//----- Callbacks to begin and end translator "session"
typedef int (IL_DECL *ILTR_PBEGINSESSION) // Pointer to begin callback
( ILXTR_PINITPARMS lpInitParms,         // Pointer to parameter block
  ILXTR_HAPPSSESSION *lpAppSession );    // Returned session handle
typedef int (IL_DECL *ILTR_PENDSESSION)   // Pointer to end callback
( ILXTR_HAPPSSESSION hAppSession );     // Session handle

/*-----
 * PRIVATE portion of translation structure. These structure members
 * are exclusively set and accessed by the translation engine.
 *
 * WARNING: do not add members to the PRIVATE portion. Even if a new
 * member SHOULD be Private, add it to the end of the ILTR_TRANSL struct
 * anyway, to avoid incompatible changes in struct member offsets!!
 *-----*/
typedef struct
{
    ILTB_ID nFilterID;                    // Filter ID
    INT16 nFldErrorNum;                   // Number of log errors
    ILTB_ID nMapID;                       // Field map ID
    INT16 nProcessStage;                  // Processing stage
    ILTB_ID nSourceID;                    // Source application ID
    ILTB_ID nTargetID;                    // Target application ID
    long nRecords;                        // Total number of records
    long recNum;                          // Current record number
    long nSrcTime;                        // Source file timestamp
    long nTarTime;                        // Target file timestamp
    char szLineTerm[ILTR_MAX_TERM];       // Line terminator string
    char szLogFile[ILTB_MAX_PATH];        // Log file name
    ILX32_16PAD(pad_01,195)              // V16: Long file names
    char szRecName[MAX_MSG];              // Current record name
    char szWorkFile[ILTB_MAX_PATH];       // Intermediate file name
    ILX32_16PAD(pad_02,195)              // V16: Long file names
    void IL_DIST * appData;               // Pointer to application data
    BOOL16 bFilter;                       // Are we called for filtering?
    INT16 nFldError;                      // Number of field errors

```



```

    BOOL16 cpack;                // Are we running under CPACK?
    BOOL16 direction;            // Importing or exporting?
    BOOL16 ebi;                  // Export before import?
    BOOL16 log;                   // Create log file?
    BOOL16 reversed;             // Source and target reversed?
    HILIF view;                   // Intermediate file view
    IL_HANDLE hAppData;          // Handle to application data
    ILX32_16PAD(pad_03,2)        // Padding for tr purposes
    IL_HFILE hLog;                // Log file handle
    ILX32_16PAD(pad_04,2)        // Padding for tr purposes
    ILDF_HNDL hFlt;              // Handle to Filters table
    ILDF_HNDL hFld;              // Handle to Fields table
    ILST_HNDL hRes;              // Resource file handle
    ILX32_16PAD(pad_05,2)        // Padding for tr purposes
    ILTR_ACTION action;          // Action type
    ILTR_FLDMAP map;              // Field map table
    ILTR_BUFFER rec;             // Intermediate record buffer
    ILTR_BUFFER field;           // Field buffer
    ILTR_FLDERR fldError[ILTR_MAX_FLDERR]; // Field errors
    ILTR_FLDLST list;            // Pointer to field names
    ILTR_CBFUN cbBegin;          // Pointer to Begin callback
    ILTR_CBFUN cbEnd;            // Pointer to End callback
    ILTR_CBFUN cbGet;            // Pointer to Get callback
    ILTR_CBFUN cbNew;            // Pointer to New callback
    ILTR_CBFUN cbPut;            // Pointer to Put callback
    ILTR_CBREC cbRecon;          // Pointer to Reconcile callback
    ILTR_CBREP cbRepeat;         // Pointer to Repeat callback
    #ifdef SYSMGR                 // System Manager definition
        IL_PSTR farPtrTable[MAX_FAR_PTRS]; // Far pointer table
    #endif // SYSMGR
    INT16 rc;                     // Translator return code
    IL_HWIN hProgWin;            // Progress window handle
    ILX32_16PAD(pad_06,2)        // Padding for tr purposes
    IL_HWIN hFromBarWin;         // FROM bar window handle
    ILX32_16PAD(pad_07,2)        // Padding for tr purposes
    IL_HWIN hToBarWin;           // TO bar window handle
    ILX32_16PAD(pad_08,2)        // Padding for tr purposes
    ILCR_LPDATA recon;           // Reconciliation data
    ILTR_PDRIVER pDriver;        // Pointer to translator driver
    UINT16 hSessionID;           // Session handle
    INT16 nTimerRecs;            // Records processed per timer
    ILTB_ID nSrcSection;         // Source section ID
    ILTB_ID nTarSection;         // Target section ID
    ILTB_ID nSourceImportCharMapID; // Char map table ID for import
    ILTB_ID nSourceExportCharMapID; // Char map table ID for export
    ILTR_BUFFER sExportCharMap;  // Export char map buffer
    /*-----
    * WARNING: do not add members to ILTR_PRIVATE. Even if a new
    * member SHOULD be Private, add it to the end of the ILTR_TRANSL struct
    * anyway, to avoid incompatible changes in struct member offsets!!
    *-----*/
} ILTR_PRIVATE;

/*-----
* Translation information structure. This structure contains
* the detailed translation parameters and is accessed by both
* the translation engine and individual translators. The
* translators should only reference the PUBLIC structure
* members and not those contained in the PRIVATE portion of
* the structure.
*-----*/
typedef struct _transl
{
    //----- PUBLIC structure members
    ILTB_CLASS nSysClass;         // System class
    ILTB_TYPE nSysType;           // System type
    long nDate;                   // Encoded current date
    unsigned long nAttribs;       // Application attributes
    char CDFsep;                  // CDF field separator
    char szCurWD[ILTB_MAX_DIR];  // Working directory
    ILX32_16PAD(pad_01,191)       // V16: Long file names
    char szAppFile[ILTB_MAX_PATH]; // Application file name
    ILX32_16PAD(pad_02,195)       // V16: Long file names
    char szAppName[MAX_APP_NAME]; // Application name

```



```

char szPswd[ILTR_MAX_PSWD];           // File password
char szSectName[MAX_APP_NAME];        // Section or category name
char szSectCode[MAX_APP_NAME];        // Section code
BOOL16 nFileExists;                   // Does file exist?
BOOL16 CDFmapOnly;                     // Use only mapped fields?
BOOL16 CDFnames;                       // Names as first CDF record?
ILTR_DIRECTION nCmd;                  // Import or Export?
ILTR_FUNCTION nFunction;               // Function type
ILTR_RANGE nSchOpt;                    // Schedule option
ILTR_UPDOPT nUpdOpt;                   // Update option
INT16 nPass;                           // Pass number
long nLoDate;                          // Earliest appointment date
long nHiDate;                          // Latest appointment date
char szAltApp[MAX_APP_NAME];           // Application name
char szAltSect[MAX_APP_NAME];          // Section or category name
char szAppLoc[ILTB_MAX_PATH];          // Application location
ILX32_16PAD(pad_03,195)                // V16: Long file names
IL_HINST hParentInst;                  // Parent instance handle
ILX32_16PAD(pad_04,2)                   // Padding for tr purposes
IL_HWIN hParentWin;                     // Parent window handle
ILX32_16PAD(pad_05,2)                   // Padding for tr purposes

/*-----
 * PRIVATE portion of structure used exclusively by the translation
 * engine and inaccessible to data translators.
 *-----*/
ILTR_PRIVATE Private;
//----- next member used to be called cbSynchroCleanup
ILUT_PBUFFER pFanBuf;                  // V26: reusable buf for repeat.c
ILXTR_SYNC_OPTION nSynchronize;        // short int; see ilxtr.h
unsigned long nSectionAttribs;          // Section attributes
INT16 nSystemError;                    // System-specific error code
BOOL16 bMultiSession;                  // TRUE for multiSession option
IL_CBPROGRESS cbProgress;              // Progress-reporting callback
UINT16 uTimerInterval;                 // milliseconds (ZERO = 55)
UINT16 uWaitInterval;                  // milliseconds (ZERO = 55)
BOOL16 bInitDone;                      // Was translator initialized?
ILTR_ENV eEnvironment;                 // Help environment
UINT16 uDlgProgFlags;                  // flags used in 'dlgprog.c'
HFONT hProgressBarFont;                // font used in 'dlgprog.c'
ILX32_16PAD(pad_06,2)                   // Padding for tr purposes
ILIF_PGLOBS pILIF_Globals;             // ptr to ILIF 'globals' struct
ILT_PILTIF pILTIF;                     // pointer to ILTIF data structure
long lImportRecs;                       // Number of imported records
char bExpInitialized;                  // Flag to init Export only once
char bImpInitialized;                  // Flag to init Import only once
INT16 version;                          // set to ILTR_CURRENT_VERSION
// by engine.

/*-----
 * for each new member, comments indicate version number assigned at
 * time of new member introduction:
 *-----*/
char phase;                             // V02: use ILTR_PHASE_XXX #defines
IL_HANDLE hILIF_Globals;                 // V03: handle of ILIF 'globals' struct
ILX32_16PAD(pad_07,2)                   // V03: Padding for tr purposes

/*-----
 * Help information.
 *-----*/
UINT32 nHelpContext;                    // V04: Help context number (4)
/*-----
 * WARNING: accesses beyond this point will GPF when running under
 *          version 3.22 or earlier of ILWIN or ILX.
 *-----*/
char szHelpFile[ILTB_MAX_PATH];          // V04: HELP file name (65)
ILX32_16PAD(pad_08,195)                 // V16: Long file names
ILX_BOOL bvWRHelp;                       // V04: Use VWR help? (2)

long lExportRecs;                        // V05: Number of exported records

/*-----
 * Translators base names.
 *-----*/
char szSrcTrans[ILTB_MAX_DRVNAME];       // V06: Source Translator Name (9)
char szTarTrans[ILTB_MAX_DRVNAME];       // V06: Target Translator Name (9)

```

```

//----- Operating environment
ILTR_ENV nRunEnviron;           // V07: Execution environment
UINT32 CRPolicy;                // V08: Conflict Resolution Policy
                                // (ILXTR_CR_POLICY_XXX)

/*-----
 * Formerly used for "internal" ILIniTranslator function pointer. This
 * pointer is no longer used, but must be left as a NULL pointer so that
 * translators using an older version of the tr structure won't try to
 * call ILIniTranslator using this pointer.
 *-----*/
FARPROC pMustBeNullPtr;         // V09: Support "internal" xlators

//----- pointer to translator extra data passed by engine
void * pXtraData;               // V10: additional translator data

/*-----
 * Next flag is used by dlgprog.c, import.c, and export.c. Says whether
 * we need to do End Processing to clean up before quitting.
 *-----*/
BOOL16 bMustCleanUpBeforeQuitting; // V11:
ILTR_BUFFER FilterFieldBuffer;     // V12: replaces "_filt" global
IL_HANDLE hxFilter;                // V12: handle used in filters.c
ILX32_16PAD(pad_09,2)              // V12: Padding for tr purposes
ILTR_PFILTER pxFilter;              // V12: pointer used in filters.c
UINT32 Flags;                      // V13: use ILTR_FLAG_XXX #defines

/*-----
 * The following handle/pointer locates table information preloaded by
 * the engine and used by ILSetupFieldLists (in LOADFLDS.C) to setup the
 * field list, field mapping and character mapping information in the tr
 * structure. If these values have NOT been set, the missing information
 * will be read from TABLES.ITB for EACH import or export phase using the
 * original IILFldLoadMap function (in LOADMAP.C).
 *-----*/
IL_HANDLE      hTableInfo;          // V14: handle for ITB table info
ILX32_16PAD    (pad_10,2)          // V14: Padding for tr purposes
ILTR_PTABLEINFO pTableInfo;        // V14: pointer to ITB table info
UINT8          TodoPriorityType;    // V15: use ILTR_TODO_PRIORITY_XXX
                                // V16: padding changes were made
INT16          nSubSectionType;     // V17: sub-section type
ILUT_PBUFFER   pTmpBuf;            // V18: reusable TEMP buffer
IL_HANDLE      hTmpBuf;            // V19: handle for TEMP buffer
ILX32_16PAD    (pad_11,2)          // V19: Padding for tr purposes
ILTM_DTTM_FMT  DtTmFmt;            // V20: Date&Time Format Params
ILUT_PBUFFER   pSSTBuf;            // V21: reusable buf for sst.c to use
IL_HANDLE      hSSTBuf;            // V21: handle for SST buffer
ILX32_16PAD    (pad_12,2)          // V21: Padding for tr purposes
BYTE           SourceSST;           // V21: Source Section SubType
BYTE           TargetSST;           // V21: current Target SectionSubType
ILTR_PSSTLIST  pTargetSSTS;        // V21: List of ALL Target SSTs
INT16          TifPhase;            // V22: phase of TIF operation
long           lEarliestDate;       // V23: used in EXPORT.C
long           lLatestDate;         // V23: used in EXPORT.C
char           cDateType;           // V23: Date Field Type (EXPORT.C)

//----- Translation session handles
ILXTR_HAPPSESSION hAppSession;     // V24: App session handle

/*-----
 * WARNING: Starting after TR variable hAppSession, all new
 * variables in the TR structure will not be visible to
 * translators under ILWIN 3.41 or before. This is because
 * the prior size of TR has been exceeded and is not
 * allocated in these prior versions. This is also true of
 * releases of Lite shipped in 1995.
 *-----*/
long           lFanningMinDate;     // V25: lower limit (0 for unlimited)
long           lFanningMaxDate;     // V25: upper limit (0 for unlimited)
ILTR_FANOUT_MAXIMA FanoutMaxima;    // V26: struct defined in ILRPT.H
BOOL16         bCancelPending;      // V27: let procmgs know about pending cancel
UINT32         OKTP_Threshold;      // V28: OKToProceed dialog threshold

//----- User directory name (or default such as "ILDATA")

```

```

char szUserDir [ILX_MAX_USERDIR];    // V29: User directory name

//----- Target system character map buffer and table IDs
ILTR_BUFFER sImportCharMap;          // V30:
ILTB_ID nTargetImportCharMapID;      // V30:
ILTB_ID nTargetExportCharMapID;      // V30:

/*-----
 * Whenever you add a new member to this structure definition, you must
 * do the following things:
 *
 * 1. change the "#define ILTR_CURRENT_VERSION <nnn>" line found at
 *    the top of this include file. Increment <nnn> by 1.
 *
 * 2. include new version number in comment for new member(s).
 *
 * 3. OBSOLETE: [reduce ILTR_MAX_FREE] we no longer do this
 *
 * 4. add a #define ILTR_newMember (tr->newMember) line at the end
 *    of the list of access macros for PUBLIC members, below.
 *
 * 5. if the size of the new member will be different between WIN16
 *    and WIN32 add an additional line directly under the member
 *    added. The line will contain the following
 *
 *        ILX32_16PAD(pad_xx,yy)                // Padding for tr purposes
 *
 *    where xx is the last pad name incremented by one, and yy is
 *    the absolute difference between the size of the new member
 *    under WIN16 and its size under WIN32. This is to maintain
 *    compatible sizes between WIN32 and WIN16 (See note in ilmacro.h
 *    for details).
 *
 * 6. if it is necessary for the ILX16 value of the new member
 *    to be passed back up to ILX32, you must add code to the
 *    ILX_V3\loadxltr.c\Update_tr_AfterILX16 function.
 *-----*/

char filler[200];                    // Free space -- DO NOT REDUCE!!
/*-----
 * Always keep 200 "slack" bytes at the end of the "tr" structure.
 * Do NOT reduce this value when the "tr" structure grows. We always want
 * a generous amount of slack so that newer translators can run under older
 * engines. We used to shrink the "slack" area over time, but that works
 * to our disadvantage. So don't do it!!
 * NOTE: there is nothing magic about the number 200. We may want to use
 * an even bigger slack area some day...
 *-----*/

} ILTR_TRANSL;

/*-----
 * Today, 2/10/95, packed structure sizes are: DJB
 *
 * WIN16:  sizeof(ILTR_TRANSL) = 1463
 * WIN32:  sizeof(ILTR_TRANSL) = 1523
 *
 * Today, 2/24/95, packed structure sizes are: KPD
 *
 * WIN16:  sizeof(ILTR_TRANSL)(ILX32_16 defined)    = 1503
 * WIN16:  sizeof(ILTR_TRANSL)(ILX32_16 not defined) = 1463
 * WIN32:  sizeof(ILTR_TRANSL)(ILX32_16 not defined) = 1503
 *-----*/

//----- Schedule item used for data reconciliation
typedef struct
{
    long date;                // Relative date
    long stTime;              // Relative start time
    long endTime;             // Relative stop time
    IL_PSTR desc;             // Appointment description
} ILTR_SCHITEM;

```

```

//----- Schedule entry for data reconciliation
typedef struct
{
    IL_PSTR importType;           // Dialog title
    INT16 hhsz;                   // Max size of handheld entry
    ILTR_SCHITEM oId;             // Desktop schedule item
    ILTR_SCHITEM newone;          // Handheld schedule item
} ILTR_SCHNOTIFY;

//----- Telephone entry for data reconciliation
typedef struct
{
    IL_PSTR importType;           // Dialog title
    INT16 fldsize;                // Max size of data
    IL_PSTR keyfname;             // Key field name
    IL_PSTR keydata;              // Key field value
    IL_PSTR pcfname;              // Desktop field name
    IL_PSTR pcdata;               // Desktop field value
    IL_PSTR hhfname;              // Handheld field name
    IL_PSTR hhdata;               // Handheld field value
} ILTR_TELNOTIFY;

//----- Globals.
extern IL_HINST hXlatorInst;     // Translator DLL instance

//----- Macros.

//----- Locate field node in field list
#define FIND_FIELD_NODE(a, b, c) \
{ \
    b = a; \
    while (b) \
    { \
        if (b->Index == c) \
            break; \
        else b = b->NextField; \
    } \
}

//----- Determine total number of fields in list
#define ILFldCount(tr) \
    tr->Private.list.Count

//----- Return name of view field
#define ILGetKeyName(tr, name, len) \
{ \
    name[0] = '\0'; \
    if (tr->Private.direction == ILTR_IMPORT) \
    { \
        if (tr->Private.map.ShowTarget != -1) \
            ILFldName(tr, tr->Private.map.ShowTarget, name, len); \
    } \
    else if (tr->Private.map.ShowSource != -1) \
        ILFldName(tr, tr->Private.map.ShowSource, name, len); \
}

//----- Register Begin callback routine
#define ILRegisterBeginCB(tr, func) \
    tr->Private.cbBegin = (ILTR_CBFUN) func

//----- Register End callback routine
#define ILRegisterEndCB(tr, func) \
    tr->Private.cbEnd = (ILTR_CBFUN) func

//----- Register Get callback routine
#define ILRegisterGetCB(tr, func) \
    tr->Private.cbGet = (ILTR_CBFUN) func

//----- Register New callback routine
#define ILRegisterNewCB(tr, func) \
    tr->Private.cbNew = (ILTR_CBFUN) func

//----- Register Put callback routine
#define ILRegisterPutCB(tr, func) \

```

```

    tr->Private.cbPut = (ILTR_CBFUN) func

//----- Register Reconcile callback routine
#define ILRegisterReconCB(tr, func) \
    tr->Private.cbRecon = (ILTR_CBREC) func

//----- Register Repeat callback routine
#define ILRegisterRepeatCB(tr, func) \
    tr->Private.cbRepeat = (ILTR_CBREP) func

//----- Specify action taken on current item
#define ILSetAction(tr, act) \
    tr->Private.action = act

//----- Set the application line terminator character
#define ILSetLineTerm(tr, term) \
{ \
    tr->Private.szLineTerm[0] = '\0'; \
    IL_STRNCAT (tr->Private.szLineTerm, term, ILTR_MAX_TERM-1); \
}

//----- Set the name of section or category
#define ILSetSectName(tr, name) \
{ \
    tr->szSectName[0] = '\0'; \
    IL_STRNCAT (tr->szSectName, name, MAX_APP_NAME); \
}

//----- Set record count
#define ILSetRecCount(tr, num) \
{ \
    ILStatusSetRecCount ((tr), ((long) (num))); \
}

//----- Set the record title
#define ILSetRecName(tr, name) \
{ \
    tr->Private.szRecName[0] = '\0'; \
    IL_STRNCAT (tr->Private.szRecName, name, MAX_MSG-1); \
}

//----- Specify number of records processed per timer event
#define ILSetRecsPerTimer(tr,num) \
{ \
    tr->Private.nTimerRecs = (num > 0 ? num : 1); \
}

//----- Set # of milliseconds to wait durring ILTR_ERR_WAIT
#define ILSetWaitInterval(tr, uTime) \
    tr->uWaitInterval = uTime

/*-----
 * Access macros to PUBLIC members of ILTR_TRANSL structure.
 * All macros assumes that the ILTR_TRANSL structure is named "tr".
 *-----*/
#define ILTR_CDFmapOnly (tr->CDFmapOnly)
#define ILTR_CDFnames (tr->CDFnames)
#define ILTR_CDFsep (tr->CDFsep)
#define ILTR_hParentWin (tr->hParentWin)
#define ILTR_hParentInst (tr->hParentInst)
#define ILTR_nAttribs (tr->nAttribs)
#define ILTR_nCmd (tr->nCmd)
#define ILTR_nDate (tr->nDate)
#define ILTR_nFileExists (tr->nFileExists)
#define ILTR_nFunction (tr->nFunction)
#define ILTR_nHiDate (tr->nHiDate)
#define ILTR_lDateRangeEnd (tr->nHiDate)
#define ILTR_nLoDate (tr->nLoDate)
#define ILTR_lDateRangeStart (tr->nLoDate)
#define ILTR_nPass (tr->nPass)
#define ILTR_nSchOpt (tr->nSchOpt)
#define ILTR_nSectionAttribs (tr->nSectionAttribs)
#define ILTR_nSysClass (tr->nSysClass)
#define ILTR_nSystemError (tr->nSystemError)
#define ILTR_nSysType (tr->nSysType)

```

```

#define ILTR_nUpdOpt          (tr->nUpdOpt)
#define ILTR_szAltApp         (tr->szAltApp)
#define ILTR_szAltSect       (tr->szAltSect)
#define ILTR_szAppFile       (tr->szAppFile)
#define ILTR_szAppLoc        (tr->szAppLoc)
#define ILTR_szAppName       (tr->szAppName)
#define ILTR_szCurWD         (tr->szCurWD)
#define ILTR_szPswd          (tr->szPswd)
#define ILTR_szSectCode      (tr->szSectCode)
#define ILTR_szSectName      (tr->szSectName)
#define ILTR_pFanBuf         (tr->pFanBuf)
#define ILTR_nSynchronize    (tr->nSynchronize)
#define ILTR_bMultiSession   (tr->bMultiSession)
#define ILTR_cbProgress      (tr->cbProgress)
#define ILTR_uTimerInterval  (tr->uTimerInterval)
#define ILTR_uWaitInterval   (tr->uWaitInterval)
#define ILTR_bInitDone       (tr->bInitDone)
#define ILTR_eEnvironment    (tr->eEnvironment)
#define ILTR_uDlgProgFlags   (tr->uDlgProgFlags)
#define ILTR_hProgressBarFont (tr->hProgressBarFont)
#define ILTR_pILIF_Globals   (tr->pILIF_Globals)
#define ILTR_pILTIF          (tr->pILTIF)
#define ILTR_lImportRecs     (tr->lImportRecs)
#define ILTR_bExpInitialized  (tr->bExpInitialized)
#define ILTR_bImpInitialized  (tr->bImpInitialized)
#define ILTR_version         (tr->version)
#define ILTR_phase           (tr->phase)
#define ILTR_hILIF_Globals   (tr->hILIF_Globals)
#define ILTR_nHelpContext    (tr->nHelpContext)
#define ILTR_szHelpFile      (tr->szHelpFile)
#define ILTR_bVWRHelp        (tr->bVWRHelp)
#define ILTR_lExportRecs     (tr->lExportRecs)
#define ILTR_szSrcTrans       (tr->szSrcTrans)
#define ILTR_szTarTrans       (tr->szTarTrans)
#define ILTR_nRunEnviron     (tr->nRunEnviron)
#define ILTR_CRPolicy         (tr->CRPolicy)
#define ILTR_pMustBeNullPtr   (tr->pMustBeNullPtr)
#define ILTR_pXtraData        (tr->pXtraData)
#define ILTR_bMustCleanUpBeforeQuitting \
    (tr->bMustCleanUpBeforeQuitting)
#define ILTR_FilterFieldBuffer (tr->FilterFieldBuffer)
#define ILTR_hxFilter          (tr->hxFilter)
#define ILTR_pxFilter          (tr->pxFilter)
#define ILTR_Flags             (tr->Flags)
#define ILTR_hTableInfo        (tr->hTableInfo)
#define ILTR_pTableInfo        (tr->pTableInfo)
#define ILTR_TodoPriorityType   (tr->TodoPriorityType)
#define ILTR_nSubSectionType   (tr->nSubSectionType)
#define ILTR_pTmpBuf           (tr->pTmpBuf)
#define ILTR_hTmpBuf           (tr->hTmpBuf)
#define ILTR_DtTmFmt           (tr->DtTmFmt)
#define ILTR_pSSTBuf           (tr->pSSTBuf)
#define ILTR_hSSTBuf           (tr->hSSTBuf)
#define ILTR_SourceSST         (tr->SourceSST)
#define ILTR_TargetSST         (tr->TargetSST)
#define ILTR_pTargetSSTS       (tr->pTargetSSTS)
#define ILTR_TargetSSTCount    (ILTR_pTargetSSTS->sstCount)
#define ILTR_TargetSSTList     (ILTR_pTargetSSTS->sstList)
#define ILTR_TifPhase          (tr->TifPhase)
#define ILTR_lEarliestDate     (tr->lEarliestDate)
#define ILTR_lLatestDate      (tr->lLatestDate)
#define ILTR_cDateType         (tr->cDateType)
#define ILTR_hAppSession       (tr->hAppSession)
#define ILTR_lFanningMinDate   (tr->lFanningMinDate)
#define ILTR_lFanningMaxDate   (tr->lFanningMaxDate)
#define ILTR_FanoutMaxima      (tr->FanoutMaxima)
#define ILTR_nDailyMaxFanout   (ILTR_FanoutMaxima.nDailyMaxFanout)
#define ILTR_nWeeklyMaxFanout  (ILTR_FanoutMaxima.nWeeklyMaxFanout)
#define ILTR_nMonthlyMaxFanout (ILTR_FanoutMaxima.nMonthlyMaxFanout)
#define ILTR_nQuarterlyMaxFanout (ILTR_FanoutMaxima.nQuarterlyMaxFanout)
#define ILTR_nYearlyMaxFanout  (ILTR_FanoutMaxima.nYearlyMaxFanout)
#define ILTR_nOtherMaxFanout   (ILTR_FanoutMaxima.nOtherMaxFanout)
#define ILTR_bCancelPending    (tr->bCancelPending)
#define ILTR_OKTP_Threshold    (tr->OKTP_Threshold)
#define ILTR_szUserDir         (tr->szUserDir)

```

```

/*-----
 * Access macros to PRIVATE symbols in ILTR_PRIVATE structure.
 * All macros assume that the ILTR_TRANS structure is named "tr".
 *-----*/
#define ILTR_action (tr->Private.action)
#define ILTR_ILCR_action (tr->Private.recon->action)
#define ILTR_appData (tr->Private.appData)
#define ILTR_bFilter (tr->Private.bFilter)
#define ILTR_nFldError (tr->Private.nFldError)
#define ILTR_cbBegin (tr->Private.cbBegin)
#define ILTR_cbEnd (tr->Private.cbEnd)
#define ILTR_cbGet (tr->Private.cbGet)
#define ILTR_cbNew (tr->Private.cbNew)
#define ILTR_cbPut (tr->Private.cbPut)
#define ILTR_cbRecon (tr->Private.cbRecon)
#define ILTR_cbRepeat (tr->Private.cbRepeat)
#define ILTR_cpack (tr->Private.cpack)
#define ILTR_direction (tr->Private.direction)
#define ILTR_ebi (tr->Private.ebi)
#define ILTR_farPtrTable (tr->Private.farPtrTable)
#define ILTR_field (tr->Private.field)
#define ILTR_fldError (tr->Private.fldError)
#define ILTR_hAppData (tr->Private.hAppData)
#define ILTR_hFromBarWin (tr->Private.hFromBarWin)
#define ILTR_hToBarWin (tr->Private.hToBarWin)
#define ILTR_hFld (tr->Private.hFld)
#define ILTR_hFlt (tr->Private.hFlt)
#define ILTR_hLog (tr->Private.hLog)
#define ILTR_hProgWin (tr->Private.hProgWin)
#define ILTR_hRes (tr->Private.hRes)
#define ILTR_hSessionID (tr->Private.hSessionID)
#define ILTR_ILCR (tr->Private.recon)
#define ILTR_ILCR_itemID (tr->Private.recon->curItem->ndxItem)
#define ILTR_ILCR_msg (tr->Private.recon->msg)
#define ILTR_list (tr->Private.list)
#define ILTR_log (tr->Private.log)
#define ILTR_map (tr->Private.map)
#define ILTR_nFldErrorNum (tr->Private.nFldErrorNum)
#define ILTR_nMapID (tr->Private.nMapID)
#define ILTR_nFilterID (tr->Private.nFilterID)
#define ILTR_nProcessStage (tr->Private.nProcessStage)
#define ILTR_nRecords (tr->Private.nRecords)
#define ILTR_nSourceID (tr->Private.nSourceID)
#define ILTR_nSrcSection (tr->Private.nSrcSection)
#define ILTR_nSrcTime (tr->Private.nSrcTime)
#define ILTR_nTargetID (tr->Private.nTargetID)
#define ILTR_nTarSection (tr->Private.nTarSection)
#define ILTR_nTarTime (tr->Private.nTarTime)
#define ILTR_pDriver (tr->Private.pDriver)
#define ILTR_rc (tr->Private.rc)
#define ILTR_rec (tr->Private.rec)
#define ILTR_recNum (tr->Private.recNum)
#define ILTR_reversed (tr->Private.reversed)
#define ILTR_szLineTerm (tr->Private.szLineTerm)
#define ILTR_szLogFile (tr->Private.szLogFile)
#define ILTR_szRecName (tr->Private.szRecName)
#define ILTR_szWorkFile (tr->Private.szWorkFile)
#define ILTR_view (tr->Private.view)
#define ILTR_nTimerRecs (tr->Private.nTimerRecs)
#define ILTR_nSourceImportCharMapID (tr->Private.nSourceImportCharMapID)
#define ILTR_nSourceExportCharMapID (tr->Private.nSourceExportCharMapID)
#define ILTR_sExportCharMap (tr->Private.sExportCharMap)
#define ILTR_sImportCharMap (tr->Private.sImportCharMap)
#define ILTR_nTargetImportCharMapID (tr->Private.nTargetImportCharMapID)
#define ILTR_nTargetExportCharMapID (tr->Private.nTargetExportCharMapID)

#ifdef __cplusplus // special handling for C++ code
}
#endif // __cplusplus

#include "iltrfn.h" // ILTR function prototypes
#include "iltiffn.h" // TIF function prototypes

#endif // __ILTR

```



```

#ifndef ILTREN_ALREADY_INCLUDED
#define ILTREN_ALREADY_INCLUDED

/*-----
 * Name:      ILTREN.H
 * Purpose:   Function Prototypes for the IntelliLink Harness Library (ILTR)
 *
 * NOTE:    Please do NOT include this header directly.  Just include ILTR.H
 *
 * Author:   Copyright (c) IntelliLink, 1992-1995
 *-----*/

//----- Special handling for C++ code
#ifdef __cplusplus
    extern "C" {
#endif // __cplusplus

#ifdef SYSMGR

    /*-----
     * Function prototypes for STDIO routines used under SYSMGR.
     * This is only because FILEIO.H and STDIO.H are mutually exclusive.
     *-----*/
    int __cdecl sprintf(char *, const char *, ...);

#endif // SYSMGR

/*-----
 * Function prototypes for common routines.
 *-----*/
void IL_DECL ILAddBenignFieldError          // Post error against field without "tally"
    ( ILTR_PTRANSI tr,                      // Name of field
      IL_PSTR szFieldName,                  // Error ID to post
      int nErrorCode );

void IL_DECL ILAddFieldError                // Post error against field
    ( ILTR_PTRANSI tr,                      // Name of field
      IL_PSTR szFieldName,                  // Error ID to post
      int nErrorCode );

int IL_DECL ILAppendLog                    // Append message to log file
    ( IL_HFILE hLog,                       // Handle to log file
      IL_HINST hXlatorInst,                // Translator instance handle
      int nError,                          // Error ID to post
      IL_PSTR lpParm1,                     // First insertion string or NULL
      IL_PSTR lpParm2 );                  // Second insertion string or NULL

int IL_DECL ILBackupFile                   // Create backup (BAK) of file
    ( ILTR_PTRANSI tr,                     // Path name of file to backup
      IL_PSTR szFile );

BOOLEAN ILComparePhone                    // Compare phone string components
    ( IL_PSTR pszPhone1,                   // Pointer to first phone string
      IL_PSTR pszPhone2,                   // Pointer to second phone string
      int iMustMatch );                  // Component "must match" flags

/*-----
 * WARNING:  ILConvertType appends to the 'toData' buffer.  Make sure your
 *           buffer is initialized before you call ILConvertType
 *-----*/
int IL_DECL ILConvertType                  // Convert between data types
    ( ILTM_DTTM_FMT_PTR pDateTimeFormat,  // Date&Time Format Params
      char fromType,                       // "From" field type (eg. 'A')
      ILTB_ATTRIB fromFieldAttributes,     // "From" field attributes
      IL_PSTR fromData,                    // "From" field value
      unsigned int fromLen,                // "From" field length
      long fromBase,                       // "From" base value (relative)
      BOOL16 bNegativeFrom,                // "From" base sign
      char toType,                         // "To" data type (eg. 'N')
      ILTB_ATTRIB toFieldAttributes,       // "To" field attributes
      IL_PSTR toData,                      // "To" data value -- APPENDED to
      unsigned int toLen,                  // "To" data length
    );

```



```

        long toBase,
        BOOL16 bNegativeTo );
        // "To" base value (relative)
        // "To" base sign

void IL_DECL ILCTStringTrans
( IL_PSTR sSrc,
  int iLen,
  char IL_DIST *pcCharTransTbl );
// Translate string via CharMap
// String to translate (in place)
// Length to be translated
// Pointer to CharMap char. table

int IL_DECL ILDateInRange
( ILTR_PTRANSL tr,
  ILTR_PREPEAT repeat,
  long theDate );
// Determine if date is contained
// Pointer to repeat record
// Numeric date

int IL_DECL ILDumpFieldsToLog
( ILTR_PTRANSL tr,
  IL_HFILE hLog );
// Dump field mapping to log file
// Handle to log file

int IL_DECL ILEndDriver
( ILTR_PTRANSL tr,
  int rc );
// Terminate translation
// Final return code

int IL_DECL ILExpandBuffer
( IL_HANDLE IL_DIST *lpHandle,
  IL_PSTR IL_DIST *lpBuffer,
  unsigned int nIncrement,
  unsigned int IL_DIST *nTotalLen );
// Expand size of dynamic buffer
// Pointer to buffer handle
// Pointer to buffer itself
// Size increment value to use
// Total buffer size to return

int IL_DECL EXP ILExport
( ILTR_PTRANSL tr );
// Perform Export operation

int IL_DECL ILFldAssoc
( ILTR_PTRANSL tr,
  IL_PSTR Label,
  IL_PSTR AssocLabel,
  BOOLEAN IL_DIST *bPositive );
// Get label of associated field
// Current internal field label
// Associated field label (ret.)
// TRUE if association is + (ret.)

int IL_DECL ILFldDesc
( ILTR_PTRANSL tr );
// Generate ILIF field descriptors

int IL_DECL ILFldFloat
( IL_PSTR buffer,
  IL_PSTR IL_DIST *item,
  unsigned int IL_DIST *len,
  ILTR_FLDPTR fld,
  BOOLEAN bStrip );
// Look for floating item in field
// Field buffer
// Floating item string to return
// Floating item length to return
// Pointer to current field node
// Do we remove floating item?

int IL_DECL ILFldGet
( ILTR_PTRANSL tr,
  IL_PSTR name,
  IL_PSTR buffer,
  unsigned int IL_DIST *len );
// Get value of field from IF
// Field name to retrieve
// Field buffer to return
// Field length to return

int IL_DECL ILFldGetEx
( ILTR_PTRANSL tr,
  IL_PSTR lpName,
  int nWhich,
  BOOLEAN bMapCharsIfNonBinary,
  IL_PSTR lpBuffer,
  unsigned int IL_DIST *pBufLen );
// variant of ILFldGet for TIF
// -- see ILTIF.CPP
// field name
// which value to retrieve
// TRUE for Character Mapping
// buffer to put value in
// IN/OUT: bufsize/valuelen

int IL_DECL ILFldGetAssoc
( ILTR_PTRANSL tr,
  int iFld,
  int IL_DIST *piAssoc );
// Get "associated field" index
// Index of requested field
// Ptr to int for associated index

int IL_DECL ILFldGetAttribs
( ILTR_PTRANSL tr,
  int iFld,
  unsigned long IL_DIST *plAtt );
// Get field attributes
// Index of requested field
// Ptr to long to store attributes

int IL_DECL ILFldGetExtName
( ILTR_PTRANSL tr,
  int iFld,
  IL_PSTR pszExtName );
// Get field external field name
// Index of requested field
// Ptr to string for external name

```

```

int IL_DECL ILFldGetIntName
( ILTR_PTRANSL tr,
  int iFld,
  IL_PSTR pszIntName );
// Get field internal field name
// Index of requested field
// Ptr to string for internal name

int IL_DECL ILFldGetItemNo
( ILTR_PTRANSL tr,
  int iFld,
  int IL_DIST *piItemNo );
// Get field item number
// Index of requested field
// Ptr to int for item number

int IL_DECL ILFldGetPrefix
( ILTR_PTRANSL tr,
  int iFld,
  IL_PSTR pszPrefix );
// Get field prefix string
// Index of requested field
// Pointer to string for Prefix

int IL_DECL ILFldGetTerm
( ILTR_PTRANSL tr,
  int iFld,
  int IL_DIST *piTerm );
// Get field terminator character
// Index of requested field
// Ptr to int for terminator char.

int IL_DECL ILFldGetType
( ILTR_PTRANSL tr,
  int iFld,
  int IL_DIST *pcType );
// Get field data type code
// Index of requested field
// Pointer to char for data type

int IL_DECL ILFldGetTypeDesc
( ILTR_PTRANSL tr,
  int iFld,
  IL_PSTR pszTypeDesc );
// Get field TypeDesc
// Index of requested field
// Pointer to string for TypeDesc

int IL_DECL ILFldGetWidth
( ILTR_PTRANSL tr,
  int iFld,
  long IL_DIST *plWidth );
// Get field width (0 == variable)
// Index of requested field
// Pointer to long for field width

void IL_DECL ILFldInsert
( ILTR_NDX IL_DIST *top,
  ILTR_NDX ndx,
  ILTR_FLDPTR field );
// Sort field node in list
// Pointer to top of field list
// Index of field to sort
// Pointer to field node

int IL_DECL ILFldInToEx
( ILTR_PTRANSL tr,
  IL_PSTR label,
  IL_PSTR name,
  int nameLen );
// Get user field name
// Internal field name
// User field name returned
// Size of user field buffer

int IL_DECL ILFldItem
( IL_PSTR IL_DIST *buffer,
  ILTR_FLDPTR lpFld,
  ILTR_NDX ndx );
// Extract next item from field
// Field buffer
// Array of field map entries
// Index of current item

int IL_DECL ILFldList
( ILTR_PTRANSL tr );
// Create list of field names

int IL_DECL ILFldLoadMap
( ILTR_PTRANSL tr,
  ILTR_FLDMAP IL_DIST *fields,
  ILTB_ID nCharMapID );
// Load field map for category
// Pointer to field map
// ITB character map table ID

int IL_DECL ILFldLookup
( ILTR_PTRANSL tr,
  IL_PSTR szInternalFieldName,
  BOOLEAN bLookInTargetFieldList,
  ILTR_FLDPTR IL_DIST *ppFieldMapEntry );
// get Field Struct for given field
// Internal field name
// use FALSE to look in src fldlst
// OUT: ptr to field map entry

int IL_DECL ILFldMapped
( ILTR_PTRANSL tr,
  IL_PSTR szInternalFieldName,
  BOOLEAN bLookInTargetFieldList );
// Determine if field is mapped
// Internal field name
// use FALSE to look in src fldlst

BOOLEAN IL_DECL ILFldName
( ILTR_PTRANSL tr,
  int itemNo,
  IL_PSTR fldName,
  int maxLen );
// Get positional field name
// Relative field number
// Field name buffer
// Size of field name buffer

```

```

int IL_DECL ILFldNameByIndex          // Get field name by index
( ILTR_PTRANS� tr,
  int itemNo,
  IL_PSTR fldName,
  int maxLen );
// Relative field number
// Field name buffer
// Size of field name buffer

BOOLEAN IL_DECL ILFldNum              // Get positional field number
( ILTR_PTRANS� tr,
  int itemNo,
  ILTR_NDX IL_DIST *pFldNum );
// field number in mini list
// OUT: field number in full list

int IL_DECL ILFldPhone                // Locate default phone field
( ILTR_PTRANS� tr,
  IL_PSTR label );
// Name of default phone field

int IL_DECL ILFldPut                  // Place value of field in IF
( ILTR_PTRANS� tr,
  IL_PSTR name,
  IL_PSTR buffer,
  unsigned int len );
// Name of field to write
// Field value to write
// Length of field value

int IL_DECL ILFldSize                 // Get maximum size of named field
( ILTR_PTRANS� tr,
  IL_PSTR label,
  unsigned long IL_DIST *pFldSize );
// Internal field name buffer
// Returned MAXIMUM field size

IL_PSTR IL_DECL ILFldStrip            // Strip field value of blanks
( IL_PSTR buffer,
  int nSides );
// Field value to strip
// Leading and/or trailing blanks?

int IL_DECL ILFldType                 // Get field type and attributes
( ILTR_PTRANS� tr,
  IL_PSTR label,
  IL_PSTR type,
  unsigned long IL_DIST *attribs );
// Internal field name to use
// Returned field type
// Returned field attributes

int IL_DECL ILFldTypeEx               // Get field type, etc.
( ILTR_PTRANS� tr,
  IL_PSTR label,
  IL_PSTR type,
  ILTB_ATTRIB IL_DIST *attribs,
  INT32 IL_DIST *pFldSize,
  IL_PSTR pszTypeDesc );
// Internal field name to use
// Returned field type
// Returned field attributes
// Returned MAXIMUM field size
// Pointer to string for TypeDesc

int IL_DECL ILFldTypeInvertedLookup   // for upside-down & backwards use
( ILTR_PTRANS� tr,
  IL_PSTR label,
  IL_PSTR type,
  ILTB_ATTRIB IL_DIST *attribs,
  INT32 IL_DIST *pFldSize,
  IL_PSTR pszTypeDesc );
// Internal field name to use
// Returned field type
// Returned field attributes
// Returned MAXIMUM field size
// Pointer to string for TypeDesc

IL_PSTR IL_DECL ILFldZeros            // Strip field of trailing zeros
( IL_PSTR buffer );
// Field value to strip

void IL_DECL ILFreeTables             // Free field mapping tables
( ILTR_PTRANS� tr );

int IL_DECL ILGetCDField              // Get relative field from string
( IL_PSTR entry,
  int fldNo,
  IL_PSTR field,
  unsigned int IL_DIST *len,
  char sep );
// String buffer
// Relative field number to get
// Field buffer to return
// Field length to return
// Field delimiter used in string

int IL_DECL ILGetOptions              // Get options in DOS command file
( ILTR_PTRANS� tr,
  IL_PSTR wd );
// IntelliLink working directory

int IL_DECL ILGetRepeat               // Get repeat record from IF
( ILTR_PTRANS� tr,
  ILTR_PREPEAT repeat );
// Pointer to repeat record

int IL_DECL EXP ILImport              // Perform Import operation

```

```

    ( ILTR_PTRANSL tr );

int IL_DECL IInitDriver
    ( ILTR_PTRANSL tr,
      IL_PSTR dir );
    // Initialize for translation
    // IntelliLink working directory

int IL_DECL IInitTranslator
    ( ILTR_PTRANSL tr,
      void IL_DIST * IL_DIST *appData );
    // Translator INIT callback
    // Pointer to application data

BOOLEAN IL_DECL IIsBoolOK
    ( IL_PSTR boolValue );
    // Validate boolean field
    // String containing boolean value

BOOLEAN IL_DECL IIsNumberOK
    ( IL_PSTR numValue );
    // Validate numeric field
    // String containing numeric value

int IL_DECL IIsRepeat
    ( ILTR_PTRANSL tr,
      long IL_DIST *nStart,
      long IL_DIST *nStop );
    // Check if item is repeating type
    // Returned start date of item
    // Returned stop date of item

int IL_DECL IIsRepeat2
    ( ILTR_PTRANSL tr,
      ILTR_PREPEAT pRepeat );
    // Check if item is repeating type
    // Repeat structure (filled in)

int ILItemHasInstancesInDateRange
    ( ILTR_PTRANSL tr,
      long lStartRange,
      long lEndRange,
      ILTR_PREPEAT pRepeat );
    // Check for instances in range
    // lower limit of date range
    // upper limit of date range
    // Repeat structure (supplied)

int IL_DECL ILoadCharMapTbl
    ( ILTB_ID nCharMapID,
      IL_PSTR pcCharMap );
    // Load a char mapping table
    // ID of map table to load
    // Pointer to map table buffer

void IL_DECL IMakePhone
    ( IL_PSTR phone,
      IL_PSTR label,
      IL_PSTR type,
      IL_PSTR country,
      IL_PSTR area,
      IL_PSTR number,
      IL_PSTR ext,
      IL_PSTR def );
    // Create complete phone number
    // Resulting phone number field
    // Phone label
    // Phone type
    // Country code
    // Area code
    // Phone number
    // Phone extension
    // Default phone number flag ('~')

int IL_DECL IMapChars
    ( IL_PSTR pInBuf,
      INT32 lInBuf,
      IL_PSTR szOldLineTerm,
      IL_PSTR szNewLineTerm,
      IL_PSTR pcMap,
      INT32 MaxLen,
      ILUT_PBUFFER pOutput );
    // map characters in a text string
    // ptr to input string
    // length of input string
    // e.g. "\r\n" or "\xFF"
    // e.g. "\xFF" or "\r\n"
    // ptr to char map
    // max STRLEN for converted string
    // -> header for reusable buffer

int IL_DECL IPutCDField
    ( IL_PSTR entry,
      int fldNo,
      IL_PSTR field,
      unsigned int IL_DIST *len,
      char sep );
    // Put relative field in string
    // String buffer
    // Relative field number to insert
    // Field value to insert
    // Max length of string
    // Field separator used in string

int IL_DECL IPutRepeat
    ( ILTR_PTRANSL tr,
      ILTR_PREPEAT repeat,
      IL_PSTR startField,
      IL_PSTR stopField );
    // Write out repeat record in IF
    // Pointer to repeat record
    // Start date field name
    // Stop date field name

void IL_DECL IReorderFloatingFields
    ( IL_PSTR szLineTerm,
      IL_PSTR szOriginal,
      IL_PSTR szInField,
      IL_PSTR szOutField );
    // Reorder field by field labels
    // Multiline field line terminator
    // Original field, set label order
    // Field to be reordered by labels
    // szInField reordered by label

int IL_DECL IRepeatItem
    ( ILTR_PTRANSL tr,

```

```

    ILTR_PREPEAT repeat,          // Pointer to repeat record
    int max );                   // max fanout count for pre-V26 era

//---- the next 2 functions are in "unfold.c". The first function does
//---- Date Range and Max Count setup and buffer management,
//---- then calls the second function.

int IL_DECL ILTRUnfoldRepeat      // expand repeat pattern into array of dates
(
    ILTR_PTRANSL tr,
    ILTR_PREPEAT pRepeat,        // IN: pattern to unfold
    int nMaxItems,               // IN: pre-V26 fanout limits
    long lToday,                 // IN: today's date
    long lFanningMinDate,        // IN: lower bound of Date Range
    long lFanningMaxDate,        // IN: upper bound of Date Range
    ILTR_PFANOUT_MAXIMA pMaxima, // IN: V26 fanout limits
    ILUT_PBUFFER IL_DIST *ppFanBuf, // IN/OUT: **buffer
    int IL_DIST *pGoodCount,      // IN/OUT: # of unexcluded instances
    int IL_DIST *pGrossCount );   // IN/OUT: # of instances

int IL_DECL ILUnfoldRepeat        // expand repeat pattern into array of dates
(
    ILTR_PREPEAT pRepeat,        // IN: pattern to unfold
    int nMaxItems,               // IN: not to exceed this count (really!)
    long lToday'sDate,           // IN: Today's date
    long lDateRangeStart,        // IN: date range lower bound
    long lDateRangeEnd,          // IN: date range upper bound
    ILUT_PBUFFER pFanBuf,        // IN/OUT: ptr to header for reusable buffer
    int IL_DIST *pGoodCount,      // OUT: # of unexcluded instances generated
    int IL_DIST *pGrossCount );   // OUT: # of instances (excluded+unexcluded)

int IL_DECL ILRepeatNextDate      // Get next date to repeat item
(
    ILTR_PREPEAT repeat,        // Pointer to repeat record
    long IL_DIST *currDate,      // Pointer to current date
    long IL_DIST *lastDate );    // Pointer to last date processed

int IL_DECL ILSetupTables         // Set up field mapping tables
(
    ILTR_PTRANSL tr );

void IL_DECL ILShowProgress       // Update progress bar
(
    ILTR_PTRANSL tr );

void IL_DECL ILSplitPhone         // Separate phone number field
(
    IL_PSTR phone,              // Full phone number buffer
    IL_PSTR label,              // Returned phone label
    IL_PSTR type,               // Returned phone type
    IL_PSTR country,            // Returned country code
    IL_PSTR area,               // Returned area code
    IL_PSTR number,             // Returned phone number
    IL_PSTR ext,                // Returned phone extension
    IL_PSTR def );              // Returned default phone flag

int IL_DECL ILTRGetField          // Get field from ILIF or from TIF
(
    ILTR_PTRANSL tr,            // tr
    IL_PSTR lpszFieldName,       // field name
    IL_PANY lpszFieldValue,      // field value
    long IL_DIST *pFieldLength ); // OUT: length; (see fldget.c)

int IL_DECL ILTRGetFieldEx        // Get field from ILIF or from TIF
(
    ILTR_PTRANSL tr,            // tr
    IL_PSTR lpszFieldName,       // field name
    int nWhich,                  // TIF_AUTO or other selector
    IL_PANY lpszFieldValue,      // field value
    long IL_DIST *pFieldLength ); // OUT: length; (see fldget.c)

/*-----
* Name:      ILTRPutField
* Purpose:   Low-level function to output a field to ILIF or TIF
*
* 'bSpecialFanningAdjustment' is TRUE when REPEAT.C UpdateDateField
*           function calls this function.
*
* NOTE:     this function may or may not do character mapping, and may or
*           may not pass data on to the FILTERS mechanism, depending on last
*           two boolean arguments that it takes.
*
*           Of course for environments that don't support FILTERS, nothing

```

```

*      is passed on to the FILTERS mechanism.
*
*      The reusable buffer 'ILTR_pTmpBuf' is used when ILTIFPutField
*      calls ILMaCharsToIL.
*
* Called from: fldput.c & repeat.c & putrep.c
*-----*/
int IL_DECL ILTRPutField ( ILTR_PTRANSL tr,
                          IL_PSTR name,
                          IL_PSTR text,
                          INT32 len,
                          BOOLEAN bSpecialFanningAdjustment,
                          BOOLEAN bDoCharacterMapping,
                          BOOLEAN bSupplyDataToFiltersMechanism );

IL_PSTR IL_DECL ILTR_TempFileName          // generate appropriate TMP file name
( int nSectionType,                        // ILTR_nFunction
  IL_PSTR lpszBuffer );                   // buffer where pathname is built

// field length (see fldput.c)
int IL_DECL EXP ILWhatFields              // Return default field list
( IL_PSTR pFileName,                      // Pointer to qualified file name
  ILTB_ID nCharMapID,                     // ID of character map
  ILTB_PHNDL phTable,                     // Pointer to table handle
  IL_HANDLE IL_DIST *phList,              // Pointer to field list handle
  int *pnList,                            // Pointer to field count
  char cDelimit );                       // ASCII delimiter character

int IL_DECL EXP ILWhatFieldsEx             // Extended ILWhatfields call
( ILX_PWFParams pWFParams );             // Ptr to WhatFields Param. Block

//----- The following function is used only by ILFldGet and ILFldPut
int IL_DECL ILMaIFErrToTRerr              // Map ILIF error to ILTR error
( int nErrILIF );                       // ILIF error code to be mapped

/*-----
* The following functions have not yet been ported to the MAC
*-----*/
#ifdef ILMAC

BOOL IL_DECL EXP clb_sect                 // Display sections dialog
( IL_HWIN hDlg,                           // Window handle
  UINT message,                           // Message ID
  WPARAM wParam,                          // wParam
  LPARAM lParam );                       // lParam

int IL_DECL ILFilterField                  // Test field for filter condition
( ILTR_PTRANSL tr,                        // Filter ID
  ILTB_ID filterID,                       // User field name
  IL_PSTR ExtName,                        // Internal field name
  IL_PSTR IntName,                        // Item number within field
  int ItemNo,                             // Field value
  IL_PSTR text );

int IL_DECL ILFilterRecord                 // Test record for filter result
( ILTR_PTRANSL tr,                        // Filter ID
  ILTB_ID filterID );

void IL_DECL ILFilterCleanup              // Terminate filter processing
( ILTR_PTRANSL tr,                        // Handle to filter file
  ILDF_PHNDL hFlt );

int IL_DECL ILFilterStartup                // Initiate for filter processing
( ILTR_PTRANSL tr );

int IL_DECL ILFldParse                    // Parse field for items
( ILTR_PTRANSL tr,                        // Field name to parse
  IL_PSTR name,                           // Field value to parse
  IL_PSTR text,                           // Length of field value
  int len );

int IL_DECL ILGetSection                  // Prompt user to select section
( ILTR_PTRANSL tr,                        // Pointer to section record
  ILTR_PSECT lpSect );

```

```

int ILInitSection                // Initialize section processing
( void );

int ILAddSection                // Add new section
( IL_PSTR szName,              // Section name
  LONG lKey );                // Section key

int ILCleanupSection            // Terminate section processing
( void );

int IL_DECL ILProccessOnTimer   // Process func. per timer event
( ILTR_PTRANSL tr,            // Millisecons per timer event
  int nTimerDelay,            // Timer CALLBACK funnction
  TIMERPROC timerFunction);

int IL_DECL ILSetDateRangeDlg   // Select date range to process
( ILTR_PTRANSL tr,            // Caller instance handle
  IL_HINST hInst,            // Pointer to encoded start date
  long *stDate,              // Pointer to encoded end date
  long *endDate);

//----- Windows-specific dialog callback function
BOOL IL_DECL EXP clb_pswd      // Display password dialog
( IL_HWIN hDlg,              // Window handle
  UINT message,              // Message ID
  WPARAM wParam,            // wParam
  LPARAM lParam );           // lParam

#endif // #ifndef ILMAC

/*-----
 * The following functions are user interface and/or OS dependent support
 * functions. The functions have the same names on both Windows and the
 * Macintosh, but the implementations of these functions are OS dependent
 * and are often implemented in different source modules for each platform.
 *-----*/
int IL_DECL ILGetPassword       // Prompt user to specify password
( ILTR_PTRANSL tr,            // Pointer to password record
  ILTR_PPSWD lpPswd );

int IL_DECL ILGetUserName       // Prompt user to specify User name
( ILTR_PTRANSL tr,            // Pointer to User ID record
  ILTR_PPSWD lpUserID );

int IL_DECL ILGetUserPassword   // Prompt user to specify User password
( ILTR_PTRANSL tr,            // Pointer to User ID record
  ILTR_PPSWD lpUserID );

void IL_DECL ILNoWaiting        // Restore normal cursor
( IL_HWIN hDlg,              // Handle to parent window
  int nMsgControl );          // Control ID to update

int IL_DECL ILProcessMessages   // Process application messages
( ILTR_PTRANSL tr );

void IL_DECL ILSetProgressText  // Change text in progress bar
( ILTR_PTRANSL tr,            // Text to reflect in progress bar
  IL_PSTR text );

void IL_DECL ILStatusBegin      // Signal "begin" to progress win
( ILTR_PTRANSL tr );

void IL_DECL ILStatusEnd        // Signal "end" to progress window
( ILTR_PTRANSL tr );

void IL_DECL ILStatusExport     // Signal "export" to progress win
( ILTR_PTRANSL tr,            // Number of records to completion
  long lNumRecs );

void IL_DECL ILStatusImport     // Signal "import" to progress win
( ILTR_PTRANSL tr,            // Number of records to completion
  long lNumRecs );

void IL_DECL ILStatusDone       // Bring progress bar to 100% done

```



```

    ( ILTR_PTRANSL tr );

void IL_DECL ILStatusInit                // Initialize iltr progress bar
( ILTR_PTRANSL tr,
  IL_PSTR szTheMsg,
  long lNumRecs );
// Progress window display text
// Number of records for 100%

void IL_DECL ILStatusSetRecCount         // (Re-)set the progress bar size
( ILTR_PTRANSL tr,
  long lNumRecs );
// Number of records to completion

void IL_DECL ILStatusUpdate              // Update progress for one record
( ILTR_PTRANSL tr );

void IL_DECL ILWaiting                   // Show hourglass while waiting
( IL_HWIN hDlg,
  IL_HINST hInstance,
  int nMsgControl,
  int nMsgID,
  IL_PSTR lpStr1,
  IL_PSTR lpStr2 );
// Handle to parent window
// Parent instance handle
// Control ID to update
// Message ID to display
// First insertion string
// Second insertion string

//----- Function prototypes for reconciliation routines.
int IL_DECL ILCRBegin
( ILTR_PTRANSL,
  IL_HINST,
  BOOLEAN );

int IL_DECL ILCREnd
( ILTR_PTRANSL,
  IL_HINST );

int IL_DECL ILCRNew
( ILTR_PTRANSL,
  IL_PSTR,
  void IL_DIST * );

int IL_DECL ILCRSetConflict
( ILTR_PTRANSL,
  IL_PSTR,
  IL_PSTR,
  IL_PSTR );

int IL_DECL ILCRReconcile
( ILTR_PTRANSL,
  IL_HINST,
  void IL_DIST * IL_DIST * );

int IL_DECL ILCRCleanUp
( ILTR_PTRANSL );

int IL_DECL ILCRSetInstruct
( ILTR_PTRANSL tr,
  IL_PSTR szInstruct );

/*-----
 * The ILSST functions, in sst.c, are for Section SubType Tagging & Filtering
 *-----*/
int IL_DECL ILSST_AddTag                // ADD TAG function for TIF to call
( ILTR_PTRANSL tr,
  INT32 SizeLimit,
  INT32 IL_DIST *pLength, // length including null terminator byte
  IL_PSTR IL_DIST *ppText,
  IL_PSTR szTagDigits,
  IL_PSTR szTypeDesc );

int IL_DECL ILSST_AddTag2               // ADD TAG function for ILFldGet to call
( ILTR_PTRANSL tr,
  unsigned int bufSize,
  unsigned int IL_DIST *pLength, // length NOT including null terminator
  IL_PSTR lpBuffer,
  IL_PSTR szTagDigits,
  IL_PSTR szTypeDesc );

int IL_DECL ILSST_Filter                // filter record by sub-type

```



```

    ( ILTR_PTRANSL tr );

int IL_DECL ILSST_StripTag      // strip tag, copy into 'szTag' arg
( ILTR_PTRANSL tr,
  IL_PSTR IL_DIST *ppText,
  unsigned int IL_DIST *pLen,
  IL_PSTR szTypeDesc,
  IL_PSTR szTag );           // OUT: tag string, e.g. "9"
                             // (char szTag[ILTR_MAX_TAG_LEN])

int IL_DECL ILSST_SubTypeOK     // apply sub-type matching rules
( ILTR_PTRANSL tr,
  IL_PSTR szSubType );

/*-----
 * The ILTRTIFxxx function, in iltrtif.c, are used to avoid rev-locks
 * between various builds of translators and various builds of ILTIF.DLL.
 * All TIF entypoints added since 12/1/1995 (ILWIN 3.40) are called via
 * ILTRTIFxxx bridge functions.
 *-----*/
int IL_DECL ILTRTIFDefFieldN    // Tell TIF about a field
( ILTR_PTRANSL tr,
  ILTR_NDX fldnum,             // field# in full field list
  INT32 ExtraAttributes );

int IL_DECL ILTRTIFFanItem      // Ask TIF to Fan a recurring item
( ILTR_PTRANSL tr,
  int maxFanCount );          // max # of fanned instances to create

int IL_DECL ILTRTIFItemIsRecurring // Ask TIF whether current item is recurring
( ILTR_PTRANSL tr );

ILBASEFN_int ILTRTIFFeatureSet  // Turn TIF features on and/or off
( ILTR_PTRANSL tr,
  UINT32 ulTurnOnBits,         // bits to turn on (see iltif.h)
  UINT32 ulTurnOffBits );      // bits to turn off (see iltif.h)

/*-----
 * Macros which handle the details of how certain special types are
 * stored in the intermediate files.
 *-----*/
#define ILTR_BOOLEAN_TRUE      "1"
#define ILTR_BOOLEAN_FALSE     "0"

//----- Write out boolean value to IF
#define ILTR_PUT_BOOLEAN(tr, nm, val)
{
    if (val)
        ILFldPut (tr, nm, ILTR_BOOLEAN_TRUE, 1);
    else
        ILFldPut (tr, nm, ILTR_BOOLEAN_FALSE, 1);
}

//----- Read boolean value from IF
#define ILTR_GET_BOOLEAN(tr, nm, val)
{
    char szTemp[2];
    unsigned int nFldSize;
    nFldSize = 2;
    ILFldGet (tr, nm, szTemp, &nFldSize);
    if (strcmp (szTemp, ILTR_BOOLEAN_TRUE, 1) == 0)
        val = TRUE;
    else
        val = FALSE;
}

/*-----
 * If the ILTR library is built with "ILTIF_IN_HARNESS" defined, then
 * executables that are linked with the ILTR library will automatically
 * load the ILTIF DLL, and will refuse to run if the ILTIF DLL isn't found.
 *-----*/
#endif ILTIF_IN_HARNESS

```

```

/*-----
 * ILTR_ILTIFxxxx functions are really just ILTIFxxxx functions.
 *-----*/
#define ILTR_ILTIFHowManyRecords(a,b)      ILTIFHowManyRecords(a,b)
#define ILTR_ILTIFInitRecord(a)            ILTIFInitRecord(a)
#define ILTR_ILTIFPutRecord(a)             ILTIFPutRecord(a)
#define ILTR_ILTIFGetField(a,b,c,d,e)      ILTIFGetField(a,b,c,d,e)
#define ILTR_ILTIFGetAndCopyField(a,b,c,d,e) ILTIFGetAndCopyField(a,b,c,d,e)
#define ILTR_ILTIFPutFieldEx(a,b,c,d,e,f)  ILTIFPutFieldEx(a,b,c,d,e,f)
#define ILTR_ILTIFGetFieldAttributes(a,b,c,d,e) \
        ILTIFGetFieldAttributes(a,b,c,d,e)
#define ILTR_ILTIFReadNextRecord(a)        ILTIFReadNextRecord(a)
#define ILTR_ILTIFItemIsRecurring(a)       ILTIFItemIsRecurring(a)
#define ILTR_ILTIFFanItem(a,b)             ILTIFFanItem(a,b)
#define ILTR_ILTIFFeatureSet(a,b,c)        ILTIFFeatureSet(a,b,c)
#define ILTR_ILTIFReopenFile(a)            ILTIFReopenFile(a)
#define ILTR_ILTIFCloseFileTemporarily(a)  ILTIFCloseFileTemporarily(a)

#else

/*-----
 * If ILTR isn't built with "ILTIF_IN_HARNESS", it is a boo-boo to
 * make calls to ILTIF entrypoints!!
 *-----*/
#define ILTR_ILTIFHowManyRecords(a,b)      ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFInitRecord(a)            ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFPutRecord(a)             ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFGetField(a,b,c,d,e)      ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFGetAndCopyField(a,b,c,d,e) ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFPutFieldEx(a,b,c,d,e,f)  ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFGetFieldAttributes(a,b,c,d,e) ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFReadNextRecord(a)        ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFItemIsRecurring(a)       ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFFanItem(a,b)             ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFFeatureSet(a,b,c)        ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFReopenFile(a)            ILTR_ERR_CANT_USE_ILTIF
#define ILTR_ILTIFCloseFileTemporarily(a)  ILTR_ERR_CANT_USE_ILTIF

#endif

#ifdef __cplusplus                // special handling for C++ code
}
#endif // __cplusplus

#endif // #ifndef ILTRFN_ALREADY_INCLUDED

```

```

/*-----
* Name:      LOADMAP.C
* Purpose:   Functions to set up Field Lists, Field Mapping and Character
*            Mapping tables into appropriate "tr" structure members.  If the
*            necessary table information has already been loaded by the
*            engine (by calling ILX_LoadFieldLists), the table information we
*            need is already available in the ILTR_TABLEINFO structure.  If
*            the tr structure does not contain a ILTR_TABLEINFO structure
*            (i.e., we're running under an "old" engine), OR if the engine
*            has not called ILX_LoadFieldLists, the table information is
*            retrieved from TABLES.ITB using ILFldLoadMap.
* Functions:  ILSetupTables -- Set up tables using ILTR_TABLEINFO if possible
*            ILFreeTables --- Free table information after import or export
*            ILFldLoadMap --- Load tables from TABLES.ITB if necessary
* Authors:   Mike Blanchette, Copyright (c) IntelliLink, 1992
*            Bob Daley, Copyright (c) IntelliLink, 1995
*-----*/

```

```

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*         a "base" DLL rather than being statically linked into every
*         translator.  Please ensure that all non-static functions in this
*         module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

```

```
#include "ilxapi.h"
```

```

/*-----
* Name:      ILSetupTables
* Purpose:   Set up tables using ILTR_TABLEINFO if possible.  ILFldLoadMap is
*            called if table information have not already been loaded in tr.
* Input:     tr -- Pointer to translation structure
* Return:    SUCCESS or ILTR_ERR code
*-----*/

```

```

ILBASEFN_int ILSetupTables          // Set up tables using ILTR_TABLEINFO
( ILTR_PTRANSL tr )                // Pointer to translation structure
{
    int nRc;                        // Return code

    //----- If no table information already loaded, use ILFldLoadMap instead
    if ((ILTR_VERSION_IS_PRIOR_TO (14)) || (ILTR_pTableInfo == NULL))
    {
        ILTB_ID nCharMapID;

        //----- Set the Character Map ID for Export or Import as appropriate
        if (ILTR_nCmd == ILTR_EXPORT)
            nCharMapID = ILTR_nSourceExportCharMapID;
        else
        {
            if ((ILTR_VERSION_IS_PRIOR_TO (30)))
                nCharMapID = ILTR_nSourceImportCharMapID;
            else
                nCharMapID = ILTR_nTargetImportCharMapID;
        }

        //----- Load/initialize the field/character map structures from TABLES.ITB
        return ILFldLoadMap (tr, &ILTR_map, nCharMapID);
    }
}

```

```

/*-----
* The table information has already been loaded into ILTR_TABLEINFO.  We
* now only need to copy this information into the appropriate "tr"
* structure members and return success.  When doing a "SyncPort" the
* source and target system ID's are reversed for phases ILTR_PHASE10 and
* ILTR_PHASE40.  We need to make sure the right information is loaded for
* the current source and target for any of the ILTR phases.
*-----*/

```

```

//----- If the source and target are NOT reversed, we use the data "asis"
if (ILTR_phase != ILTR_PHASE10 && ILTR_phase != ILTR_PHASE40)
{
    //----- Copy the complete field map table and its field list pointers
    ILTR_map = ILTR_pTableInfo->sFieldMap;
}

```

```

//----- Source/target have been swapped, so we need to swap the table data
else
{
    ILTR_PFLDMAP pMap;

    //----- Get a direct pointer to the original field map table information
    pMap = &ILTR_pTableInfo->sFieldMap;

    //----- Copy source data to target and target data to source
    ILTR_map.nSource      = pMap->nTarget;      // Count of source fields
    ILTR_map.nTarget      = pMap->nSource;      // Count of target fields
    ILTR_map.nMapStatus   = pMap->nMapStatus;    // Map status indicator
    IL_STRCPY ( ILTR_map.sName,                // Template name
               pMap->sName );                  // ..
    ILTR_map.ApptDate     = pMap->ApptDate;     // Index of appt date field
    ILTR_map.ShowSource   = pMap->ShowTarget;    // Index of source "show" field
    ILTR_map.ShowTarget   = pMap->ShowSource;    // Index of target "show" field
    ILTR_map.TopSource    = pMap->TopTarget;     // Index of top source field
    ILTR_map.TopTarget    = pMap->TopSource;     // Index of top target field
    ILTR_map.hSource      = pMap->hTarget;      // Handle to source fields
    ILTR_map.hTarget      = pMap->hSource;      // Handle to target fields
    ILTR_map.pSource      = pMap->pTarget;      // Pointer to top source field
    ILTR_map.pTarget      = pMap->pSource;      // Pointer to top target field
}

//----- Copy the character map info for Export or Import as appropriate
if (ILTR_nSynchronize)
{
    if ((ILTR_phase == ILTR_PHASE10 || ILTR_phase == ILTR_PHASE30) &&
        (ILTR_VERSION_IS_AT_LEAST (30)))
    {
        ILTR_sExportCharMap = ILTR_pTableInfo->sTargetExportCharMap;
        ILTR_sImportCharMap = ILTR_pTableInfo->sTargetImportCharMap;
    }
    else
    {
        ILTR_sExportCharMap = ILTR_pTableInfo->sSourceExportCharMap;
        ILTR_sImportCharMap = ILTR_pTableInfo->sSourceImportCharMap;
    }
}
else
{
    if (ILTR_nCmd == ILTR_EXPORT || ILTR_VERSION_IS_PRIOR_TO (30))
    {
        ILTR_sExportCharMap = ILTR_pTableInfo->sSourceExportCharMap;
        ILTR_sImportCharMap = ILTR_pTableInfo->sSourceImportCharMap;
    }
    else
    {
        ILTR_sExportCharMap = ILTR_pTableInfo->sTargetExportCharMap;
        ILTR_sImportCharMap = ILTR_pTableInfo->sTargetImportCharMap;
    }
}

/*-----
 * The source and target field lists are now set. Construct a sorted list
 * of unique, INTERNAL field names along with a count of top level fields.
 * This list will be allocated by ILFldList and freed by ILFreeTables.
 *-----*/
nRc = ILFldList (tr);
if (nRc != SUCCESS)
    return nRc;

//----- All done
return SUCCESS;
}

/*-----
 * Name:      ILFreeTables
 * Purpose:   Free table information after import or export. The table
 *            information in ILTR_TABLEINFO (when available) is NOT freed.
 *            When ILTR_TABLEINFO is being used we simply clear any handles

```

```

*          and pointers to table data copied to tr by ILSetupTables.
* Input:    tr -- Pointer to translation structure
* Return:   void
*-----*/

ILBASEFN void ILFreeTables          // Free table info after import/export
( ILTR_PTRANSL tr )                // Pointer to translation structure
{
    //----- If no table information loaded, free data allocated by ILSetupTables
    if ((ILTR_VERSION_IS_PRIOR_TO (14)) || (ILTR_pTableInfo == NULL))
    {
        //----- Free the character mapping tables.
        if (ILTR_sExportCharMap.buffer)
            IL_FREE_AND_ZERO (ILTR_sExportCharMap.handle, ILTR_sExportCharMap.buffer);
        if (ILTR_sImportCharMap.buffer)
            IL_FREE_AND_ZERO (ILTR_sImportCharMap.handle, ILTR_sImportCharMap.buffer);

        //----- Free the source and target field lists.
        if (ILTR_map.pSource)
            IL_FREE_AND_ZERO (ILTR_map.hSource, ILTR_map.pSource);
        if (ILTR_map.pTarget)
            IL_FREE_AND_ZERO (ILTR_map.hTarget, ILTR_map.pTarget);

        //----- Clear selected source/target fields.
        ILTR_map.nSource = ILTR_map.nTarget = 0;
        ILTR_map.TopSource = ILTR_map.TopTarget = -1;

        //----- Free the sorted list of source field (internal) names
        if (ILTR_list.Name)
            IL_FREE_AND_ZERO (ILTR_list.handle, ILTR_list.Name);
        ILTR_list.Count = 0;

        //----- All done.
        return;
    }

    /*-----
    * The table information in the tr structure was placed there from data in
    * ILTR_TABLEINFO. We now need clear the appropriate values in the tr
    * structure to "appear" that they have been freed. Note that we actually
    * DO free the sorted list of internal source field names in ILTR_list, as
    * this list needs to be rebuilt for each individual import or export.
    *-----*/

    //----- Clear the character mapping table handle and pointer
    ILTR_sExportCharMap.handle = IL_NULL_HANDLE;
    ILTR_sExportCharMap.buffer = NULL;
    ILTR_sImportCharMap.handle = IL_NULL_HANDLE;
    ILTR_sImportCharMap.buffer = NULL;

    //----- Clear the source and target field lists handles and pointers
    ILTR_map.hSource = ILTR_map.hTarget = IL_NULL_HANDLE;
    ILTR_map.pSource = ILTR_map.pTarget = NULL;

    //----- Clear selected fields.
    ILTR_sExportCharMap.width = 0;
    ILTR_sImportCharMap.width = 0;
    ILTR_map.nSource = ILTR_map.nTarget = 0;
    ILTR_map.TopSource = ILTR_map.TopTarget = -1;

    //----- Free the sorted list of source field (internal) names
    if (ILTR_list.Name)
        IL_FREE_AND_ZERO (ILTR_list.handle, ILTR_list.Name);
    ILTR_list.Count = 0;

    //----- All done
    return;
}

/*-----
* The remaining code (ILFldLoadMap and ILFldLoadCharMap) are here only for
* compatibility with the past. If the engine we are running under was built
* with an ILTR version earlier than ILTR version 14, OR if the engine/ILWIN
* did not call ILX_LoadFieldLists, there will not be any table information in

```

```

*   the tr structure. In these cases, ILFldLoadMap is called as in the past
*   to load the necessary information from TABLES.ITB. Once we can be sure
*   that all supported versions of ILX and ILWIN are using ILX_LoadFieldLists,
*   we can delete the following code and any references to it from above.
*-----*/

//----- Internal function to load ITB character map table
static
int IL_DECL ILFldLoadCharMap          // Load field map and CharMap tables
( ILTR_PTRANS tr,                    // Pointer to translation structure
  ILTB_PHNDL phTable,                // Pointer to ITB open file handle
  ILTB_ID nCharMapID );              // ITB character map table ID

/*-----
*   Name:      ILFldLoadMap
*   Purpose:   Create and load a field map template and CharMap table.
*   Input:     tr ----- Pointer to translation structure
*              fields --- Pointer to field map table
*              CharMap -- Character map ID
*   Return:    SUCCESS, FAILURE, or error code
*   Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/

ILBASEFN_int ILFldLoadMap              // Load field map and CharMap tables
( ILTR_PTRANS tr,                      // Pointer to translation structure
  ILTR_FLDMAP IL_DIST * fields,        // Pointer to field map table
  ILTB_ID nCharMapID )                // ITB character map table ID
{
  ILTR_NDX      i;                     // Loop variable
  int           error = SUCCESS;        // Return code
  int           fieldCount;             // Field count
  int           len;                    // Record length
  int           memSize;                // Memory size
  int           ndx;                    // Field index
  int           rc = SUCCESS;           // Return code
  ILTB_ID       sourceFldID;            // field list ID of source list in file
  ILTB_ID       targetFldID;            // field list ID of target list in file
  char          sFile[MAX_PATH];        // Temporary file name
  ILTB_HNDL     hTable;                 // Handle to ILTB tables file
  IL_HANDLE     hMap;                  // Handle to field map
  IL_HANDLE     hSource;                // Handle to source field list
  IL_HANDLE     hTarget;                // Handle to target field list
  IL_HANDLE     hTemp;                 // Temporary handle
  ILTB_PMAP     pMap = NULL;            // Pointer to map record
  ILTB_PFLDLST  pSource = NULL;         // Pointer to source field list
  ILTB_PFLDLST  pTarget = NULL;         // Pointer to target field list

  //----- Clear variables.
  fields->nSource      = fields->nTarget      = 0;
  fields->ShowSource   = fields->ShowTarget = -1;
  fields->TopSource    = fields->TopTarget   = -1;
  fields->ApptDate     = -1;
  fields->pSource      = fields->pTarget     = NULL;
  fields->nMapStatus   = 0;
  fields->sName[0]     = '\0';

  //----- Locate the ITB tables file containing field and character maps
  IL_STRCAT (IL_STRCPY (sFile, ILTR_szCurWD), IL_FILESEP_STR);
  IL_STRCAT (sFile, ILX_TABLES_FILE);

  /*-----
  *   Open System Table File in UPDATE mode, except that if the file is
  *   READONLY, open it in READ mode. Most of our apps, including ILWIN,
  *   need UPDATE access, but some, e.g. IntelliLink for WinPad,
  *   need nothing more than READ access.
  *-----*/
  if (ILTBOpen (sFile, &hTable, ILTB_MODE_AS_ALLOWED))
    return ILTR_ERR_MAPFILE;

  /*-----
  *   Initialize the tr character mapping table, using the specified ITB
  *   character map ID if both specified and available.
  *-----*/
  error = ILFldLoadCharMap (tr, &hTable, nCharMapID);

```

```

if (error)
{
    error = ILTR_ERR_NOCHARMAP;
    goto errorExit;
}

/*-----
 * Find a suitable field map ID in ITB tables file if no Map ID is
 * specified in the translation structure.
 *-----*/
if (!ILTR_nMapID)
{
    if (!(ILTR_nMapID = ILTBFindMapID ( &hTable,
                                        ILTR_nSourceID,
                                        ILTR_nSrcSection,
                                        ILTR_nTargetID,
                                        ILTR_nTarSection,
                                        &ILTR_reversed )))
    {
        error = ILTR_ERR_NOMAP;
        goto errorExit;
    }
}

//----- Now obtain the map record length.
len = ILTBGetMapRecLen (&hTable, ILTR_nMapID);
if (! len)
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

//----- Allocate sufficient memory to load map record in memory.
IL_ALLOC_MEM (len, hMap, pMap);
if (pMap == NULL)
{
    error = ILTR_ERR_NOMEM;
    goto errorExit;
}

//----- Read the map record.
if (ILTBGetMapRec (&hTable, ILTR_nMapID, NULL, pMap, len))
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

//----- Check if systems are reversed in this field map.
if (ILTBGetFldAppID (&hTable, pMap->sourceList) == ILTR_nSourceID)
    ILTR_reversed = FALSE;
else
    ILTR_reversed = TRUE;

//----- Note the map name and number of mapped fields.
IL_STRCPY (fields->sName, pMap->name);
fieldCount = ILTBGetFldFieldNum (&hTable, pMap->sourceList);

//----- Set the source and target field list ID's based on order.
if (ILTR_reversed)
{
    sourceFldID = pMap->targetList;
    targetFldID = pMap->sourceList;
}
else
{
    sourceFldID = pMap->sourceList;
    targetFldID = pMap->targetList;
}

//----- Determine the size of the field list.
len = ILTBGetFldRecLen (&hTable, sourceFldID);
if (! len)
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

```



```

    }

    //----- Allocate memory required to hold all source fields.
    IL_ALLOC_MEM (len, hSource, pSource);
    if (pSource == NULL)
    {
        error = ILTR_ERR_NOMEM;
        goto errorExit;
    }

    //----- Now read the field list into memory.
    if (ILTBGetFldRec (&hTable, sourceFldID, pSource, len))
    {
        error = ILTR_ERR_BADMAP;
        goto errorExit;
    }

    //----- Post the field associations in the source fields.
    IL_SYNC_MEM (hMap, pMap);
    for (i = 0; i < fieldCount; i++)
    {
        if ((ndx = pMap->mapIndex[i]) != ILX_UNMAPPED)
        {
            if (ILTR_reversed)
                pSource->field[ndx].mapIndex = i;
            else pSource->field[i].mapIndex = ndx;
        }
    }

    /*-----
    * Now allocate memory for the source list that will actually
    * be used by the translators.
    *-----*/
    fields->nSource = pSource->fieldNum;
    memSize = sizeof (ILTR_FIELD) * fields->nSource;
    IL_ALLOC_MEM (memSize, hTemp, fields->pSource);
    fields->hSource = hTemp;
    if (fields->pSource == NULL)
    {
        error = ILTR_ERR_NOMEM;
        goto errorExit;
    }

    //----- Process all source fields.
    for (i = 0; i < fields->nSource; i++)
    {
        //----- Copy all fields from input record to output buffer.
        fields->pSource[i].ItemNo      = pSource->field[i].itemNo;
        fields->pSource[i].Type        = (char) pSource->field[i].type;
        fields->pSource[i].Width       = pSource->field[i].width;
        fields->pSource[i].Term         = pSource->field[i].delimit;
        fields->pSource[i].Attribs      = pSource->field[i].attribs;
        fields->pSource[i].MapField     = pSource->field[i].mapIndex;
        fields->pSource[i].Assoc        = pSource->field[i].assoc;
        fields->pSource[i].Index        = -1;
        fields->pSource[i].NextField    = -1;
        fields->pSource[i].PriorField   = -1;
        IL_STRCPY (fields->pSource[i].IntName, pSource->field[i].label);
        IL_STRCPY (fields->pSource[i].ExtName, pSource->field[i].name);
        IL_STRCPY (fields->pSource[i].Label, pSource->field[i].prefix);
        IL_STRCPY (fields->pSource[i].TypeDesc, pSource->field[i].typeDesc);

        //----- Locate insertion point and post next/prior field indexes.
        ILFldInsert (&fields->TopSource, i, fields->pSource);
    }

    //----- Free the source list record.
    if (pSource)
    {
        IL_FREE_MEM (hSource, pSource);
        pSource = NULL;
    }

    //----- Determine the size of the target field list.
    len = ILTBGetFldRecLen (&hTable, targetFldID);

```

```

if (! len)
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

//----- Allocate memory required to hold all target field descriptors.
IL_ALLOC_MEM (len, hTarget, pTarget);
if (pTarget == NULL)
{
    error = ILTR_ERR_NOMEM;
    goto errorExit;
}

//----- Now read target field list into memory.
if (ILTBGetFldRec (&hTable, targetFldID, pTarget, len))
{
    error = ILTR_ERR_BADMAP;
    goto errorExit;
}

//----- Post the field associations in the target fields.
IL_SYNC_MEM (hMap, pMap);
for (i = 0; i < fieldCount; i++)
{
    if ((ndx = pMap->mapIndex[i]) != ILX_UNMAPPED)
    {
        if (ILTR_reversed)
            pTarget->field[i].mapIndex = ndx;
        else pTarget->field[ndx].mapIndex = i;
    }
}

/*-----
 * Now allocate memory for the target list that will actually
 * be used by the translators.
 *-----*/
fields->nTarget = pTarget->fieldNum;
memSize = sizeof (ILTR_FIELD) * fields->nTarget;
IL_ALLOC_MEM (memSize, hTemp, fields->pTarget);
fields->hTarget = hTemp;
if (fields->pTarget == NULL)
{
    error = ILTR_ERR_NOMEM;
    goto errorExit;
}

//----- Process all source fields.
for (i = 0; i < fields->nTarget; i++)
{
    //----- Copy all fields from input record to output buffer.
    fields->pTarget[i].ItemNo      = pTarget->field[i].itemNo;
    fields->pTarget[i].Type       = (char) pTarget->field[i].type;
    fields->pTarget[i].Width      = pTarget->field[i].width;
    fields->pTarget[i].Term       = pTarget->field[i].delimit;
    fields->pTarget[i].Attribs    = pTarget->field[i].attribs;
    fields->pTarget[i].MapField   = pTarget->field[i].mapIndex;
    fields->pTarget[i].Assoc      = pTarget->field[i].assoc;
    fields->pTarget[i].Index      = -1;
    fields->pTarget[i].NextField  = -1;
    fields->pTarget[i].PriorField = -1;
    IL_STRCPY (fields->pTarget[i].IntName, pTarget->field[i].label);
    IL_STRCPY (fields->pTarget[i].ExtName, pTarget->field[i].name);
    IL_STRCPY (fields->pTarget[i].Label, pTarget->field[i].prefix);
    IL_STRCPY (fields->pTarget[i].TypeDesc, pTarget->field[i].typeDesc);

    //----- Locate insertion point and post next/prior field indexes.
    ILFldInsert (&fields->TopTarget, i, fields->pTarget);
}

/*-----
 * The source and target field lists are now set. Construct a sorted list
 * of unique, INTERNAL field names along with a count of top level fields.
 * This list will be allocated by ILFldList and freed by ILFreeTables.
 *-----*/

```

```

    error = ILFldList (tr);

errorExit:

    //----- Free dynamic memory before leaving.
    if (pMap)
        IL_FREE_MEM (hMap, pMap);
    if (pSource)
        IL_FREE_MEM (hSource, pSource);
    if (pTarget)
        IL_FREE_MEM (hTarget, pTarget);

    //----- Close the ILTB tables file.
    rc = ILTBclose (&hTable);

    //----- Return with error code (default of SUCCESS).
    return (error ? error : rc);
}

/*-----
* Name:      ILFldLoadCharMap
* Purpose:   Initialize the character mapping table. Load system-specific
*            CharMap table from ITB if one is specified and provided.
* Input:     tr ----- Pointer to translation structure
*            phTable -- Pointer to ITB open file handle
*            CharMap -- Character map ID
* Return:    SUCCESS, FAILURE, or ILTB error code
* Author:    Bob Daley, Copyright (c) IntelliLink, 1995
*-----*/
static
int IL_DECL ILFldLoadCharMap          // Load field map and CharMap tables
(
    ILTR_PTRANSL tr,                  // Pointer to translation structure
    ILTB_PHNDL phTable,               // Pointer to ITB open file handle
    ILTB_ID nCharMapID )              // ITB character map table ID
{
    int i;                            // Temporary loop/index variable
    int iRc;                          // Function return code
    int nMemSize;                     // Number of bytes to be allocated
    IL_HANDLE hCharMap;               // Handle for CharMap record buffer
    ILTB_PCHARMAPREC pCharMap = NULL; // Pointer to CharMap record buffer

    //----- Allocate memory for character mapping tables.
    nMemSize = ILTB_CHARMAP_CHARS * sizeof (char);
    IL_ALLOC_MEM (nMemSize, ILTR_sExportCharMap.handle, ILTR_sExportCharMap.buffer);
    if (ILTR_sExportCharMap.buffer == NULL)
        return ILTR_ERR_NOMEM;

    IL_ALLOC_MEM (nMemSize, ILTR_sImportCharMap.handle, ILTR_sImportCharMap.buffer);
    if (ILTR_sImportCharMap.buffer == NULL)
        return ILTR_ERR_NOMEM;

    //----- Initialize null CharMap tables in case no CharMap provided.
    for (i = 0; i < ILTB_CHARMAP_CHARS; i++)
    {
        ILTR_sExportCharMap.buffer[i] = i;
        ILTR_sImportCharMap.buffer[i] = i;
    }

    //----- If no CharMap ID provided, simply return success.
    if (nCharMapID == 0)
        return SUCCESS;

    //----- Make sure we have a valid CharMap ID.
    if (! ILTBIsCharMapRec (phTable, nCharMapID))
    {
        //----- Simply ignore error if this ITB file doesn't support CharMaps.
        iRc = ILTBLastError (phTable);
        if (iRc == ILTB_ERR_VERSION)
            return SUCCESS;

        //----- Not an "ITB file version" issue. We have a bad CharMap ID.
        return ILTR_ERR_MAPFILE;
    }
}

```

```
//----- Get the record length for this CharMap ID.
nMemSize = ILTBGetCharMapRecLen (phTable, nCharMapID);
if (nMemSize == 0)
    return ILTR_ERR_MAPFILE;

//----- Allocate memory for the FULL CharMap record.
IL_ALLOC_MEM (nMemSize, hCharMap, pCharMap);
if (pCharMap == NULL)
    return ILTR_ERR_NOMEM;

//----- Read in the complete CharMap record.
iRc = ILTBGetCharMapRec (phTable, nCharMapID, pCharMap, nMemSize);
if (iRc)
{
    iRc = ILTR_ERR_MAPFILE;
    goto CleanUpAndReturn;
}

//----- Initialize the Character Mapping tables from the CharMap record data.
for (i = 0; i < ILTB_CHARMAP_CHARS; i++)
{
    ILTR_sExportCharMap.buffer[i] = pCharMap->cChars[i];
    ILTR_sImportCharMap.buffer[i] = pCharMap->cChars[i];
}

//----- All done, indicate successful completion.
iRc = SUCCESS;

//----- Clean up and return status.
CleanUpAndReturn:
if (pCharMap)
    IL_FREE_MEM (hCharMap, pCharMap);
return iRc;
}
```

```

/*-----
* Name:      CHARMAP.C
* Part of:   the IntelliLink Translation Harness (ILTR library)
* Contents:  functions used for Character Mapping:
*
*      ILLoadCharMapTbl - Load a character mapping table from TABLES.ITB
*
*      ILMapChars      - map characters in a string from one encoding to
*                        another, using a character mapping table and a
*                        pair of OLD and NEW line terminator strings.
*
* Author:    Roger Duchesneau, Copyright (c) IntelliLink Corporation, 1994
*            Bob Daley, Copyright (c) IntelliLink Corporation, 1995
*            David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*        a "base" DLL rather than being statically linked into every
*        translator. Please ensure that all non-static functions in this
*        module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#include "ilxapi.h"                // Common translator definitions

/*-----
* Function:  ILLoadCharMapTbl
* Purpose:   Load a character mapping table from the ITB file
* Input:     nCharMapID -- the ID of the CharMap to be loaded
*            pcCharMap --- pointer to the CharMap character table
* Returns:   SUCCESS or ILTR error code
*-----*/

ILBASEFN_int ILLoadCharMapTbl      // Load CharMap table from ITB file
( ILTB_ID nCharMapID,              // ID of CharMap to be loaded
  IL_PSTR pcCharMap )              // Pointer to CharMap table to fill in
{
    int          i;                // Loop/index variable
    int          iRc;              // Function return code
    int          nMemSize;          // Number of bytes to be allocated
    IL_HANDLE    hCharMap;          // Handle for CharMap record buffer
    ILTB_PCHARMAPREC pCharMap = NULL; // Pointer to CharMap record buffer
    ILTB_HNDL    hTable;           // Handle to ITB tables file

    //----- Initialize null CharMap table in case no CharMap provided.
    for (i = 0; i < ILTB_CHARMAP_CHARS; i++)
        pcCharMap[i] = i;

    //----- If no CharMap ID provided, simply return success.
    if (nCharMapID == 0)
        return SUCCESS;

    //----- Open the ILTB file in read-only mode.
    if (ILTBOpen (ILX_TABLES_FILE, &hTable, ILTB_MODE_READ))
        return ILTR_ERR_MAPFILE;

    //----- Make sure we have a valid CharMap ID.
    if (! ILTBIsCharMapRec (&hTable, nCharMapID))
    {
        //----- Simply ignore error if this ITB file doesn't support CharMaps.
        iRc = ILTBLastError (&hTable);
        if (iRc == ILTB_ERR_VERSION)
            return SUCCESS;

        //----- Not an "ITB file version" issue. We have a bad CharMap ID.
        return ILTR_ERR_MAPFILE;
    }

    //----- Get the record length for this CharMap ID.
    nMemSize = ILTBGetCharMapRecLen (&hTable, nCharMapID);
    if (nMemSize == 0)
        return ILTR_ERR_MAPFILE;

    //----- Allocate memory for the FULL CharMap record.
    IL_ALLOC_MEM (nMemSize, hCharMap, pCharMap);

```

```

    if (pCharMap == NULL)
        return ILTR_ERR_NOMEM;

    //----- Read in the complete CharMap record.
    iRc = ILTBGetCharMapRec (&hTable, nCharMapID, pCharMap, nMemSize);
    if (iRc)
    {
        iRc = ILTR_ERR_MAPFILE;
        goto CleanUpAndReturn;
    }

    //----- Initialize the Character Mapping table from the CharMap record data.
    for (i = 0; i < ILTB_CHARMAP_CHARS; i++)
        pcCharMap[i] = pCharMap->cChars[i];

    //----- Make sure NULL maps to NULL ... otherwise we'll go crazy!!
    if (pcCharMap[0] != 0)
        iRc = ILERROR(0, ILTR_ERR_INTERNAL_ERROR);
    else
        iRc = SUCCESS;

CleanUpAndReturn:

    //----- Free any allocated memory.
    if (pCharMap)
        IL_FREE_MEM (hCharMap, pCharMap);

    //----- Close the ILTB tables file.
    if (ILTBClose (&hTable) && iRc == SUCCESS)
        iRc = ILTR_ERR_MAPFILE;

    //----- Return status.
    return iRc;
}

/*-----
* Name:      ILMaChars - map characters in a string from one encoding
*              to another.
*
* Purpose: 1. Map characters to their new value using the specified mapping
*           table.
*           2. Switch from one line terminator string to another
*
* Returns: SUCCESS
*           or ILTR_ERR_TRUNC or ILTR_ERR_NOMEM or ILTR_ERR_INTERNAL_ERROR
*
* Author:   David Boothby, Copyright (c) IntelliLink, 1995
*
* Notes:
*
*   o. Conversion is never done in place. We always read from one buffer
*      and write to another.
*
*   o. We do NOT require that the input string be null terminated. But we
*      guarantee that the output string will be null terminated.
*
*   o. We require that one of the line terminator strings (either old or new)
*      be exactly 1 character long. The other line terminator can be either
*      1 or 2 characters long.
*
*   o. The output buffer is a reusable buffer that the user is NOT expected
*      to free.
*-----*/
ILBASEFN_int ILMaChars
(
    IL_PSTR pInBuf,
    INT32 lOldLen,           // length of input string
    IL_PSTR szOldLineTerm,
    IL_PSTR szNewLineTerm,
    IL_PSTR charmap,
    INT32 MaxLen,           // max STRLEN for converted string
    ILUT_PBUFFER pOutput ) // -> header for reusable buffer
{
    int rc, rc2;
    INT32 i;

```

```

INT32 lNewLen;
int nOldTermLen = IL_STRLEN (szOldLineTerm);
int nNewTermLen = IL_STRLEN (szNewLineTerm);
int delta = nNewTermLen - nOldTermLen;
char cNewTerm0 = szNewLineTerm[0];
char cNewTerm1 = szNewLineTerm[1];
char cOldTerm0 = szOldLineTerm[0];
char cOldTerm1 = szOldLineTerm[1];
UINT8 ucOldTerm0 = ((UINT8 *) szOldLineTerm)[0];
UINT8 ucOldTerm1 = ((UINT8 *) szOldLineTerm)[1];
IL_PSTR pOut;
UINT8 *pIn = (UINT8 *) pInBuf;
char cMapSave;

/*-----
 * Enforce restrictions:  require that both old and new line terminator
 * strings be 1 or 2 chars long, and don't allow BOTH to be 2 chars
 * long.
 *-----*/
if ( (nOldTermLen < 1)
    || (nOldTermLen > 2)
    || (nNewTermLen < 1)
    || (nNewTermLen > 2)
    || (nOldTermLen + nNewTermLen > 3) )
    return ILERROR (nOldTermLen + (10*nNewTermLen), ILTR_ERR_INTERNAL_ERROR);

/*-----
 * If conversion is other than 1-for-1 character replacement, find out
 * how many line terminators there are.
 *-----*/
if (delta != 0)
{
    int deltaCount = 0;  // number of size-changing replacements
    if (nOldTermLen == 1)
    {
        for (i=0; i < lOldLen; i++)
            if (pInBuf[i] == cOldTerm0)
                deltaCount++;
    }
    else
    {
        for (i=0; i < lOldLen-1; i++)
            if (pInBuf[i] == cOldTerm0 && pInBuf[i+1] == cOldTerm1)
            {
                deltaCount++;
                i++;          // double-step
            }
    }

    //---- See how much size will change if all replacements are made
    delta *= deltaCount;
}

/*-----
 * Compute how long converted string will be.  Don't let it exceed max.
 *-----*/
lNewLen = lOldLen + delta;
if (lNewLen <= MaxLen)
    rc = SUCCESS;
else
{
    lNewLen = MaxLen;
    rc = ILTR_ERR_TRUNC;
}

/*-----
 * Ensure that output buffer is big enough.  If not, expand it.
 *-----*/
rc2 = ILUT_GetBuffer (pOutput, lNewLen + 1);  // room for null
if (rc2 == ILUT_ERR_NOMEM)
    return ILTR_ERR_NOMEM;

else if (rc2 == ILUT_ERR_SETTLE_FOR_LESS)
{
    lNewLen = pOutput->lBufSize - 1;          // leave room for null

```



```

    rc = ILTR_ERR_TRUNC;
}

//--- get local pointer to output buffer
pOut = pOutput->pBuffer;

/*-----
 * If no size-changing replacements are required, do a simple 1-for-1
 * character replacement.
 *
 * NOTE:  this clause is used not only for conversions with same-size
 *        line terminators, but also for all conversions where the
 *        input string contains no line terminators.
 *-----*/
if (delta == 0)
{
    //--- temporarily hack charmap to map EOLs correctly
    cMapSave = charmap[ucOldTerm0];
    charmap[ucOldTerm0] = szNewLineTerm[0];

    /*-----
     * Translate string, but not more than output buffer can handle!!
     *-----*/
    for (i=0; i < lNewLen; i++)
        *pOut++ = charmap[*pIn++];

    *pOut = 0;    //--- null-terminate output string

    /*-----
     * Restore the char map to the way it was.
     *-----*/
    charmap[ucOldTerm0] = cMapSave;
}

else
{
    /*-----
     * Do size-changing substitution.  Make 'pOutMax' point to where
     * the null terminator for the output string should go.
     *-----*/
    UINT8 uIn;
    IL_PSTR pOutMax = &pOut[lNewLen];

    while ((uIn = *pIn++) != 0)
    {
        if (uIn == ucOldTerm0)
        {
            //--- looks like we hit a line terminator in the input string
            if (nOldTermLen == 1)
            {
                /*-----
                 * Put 2-char line terminator in place of 1-char terminator.
                 *-----*/
                *pOut++ = cNewTerm0;
                if (pOut == pOutMax)
                {
                    *(--pOut) = 0; // never put out HALF a line terminator
                    break;
                }
                *pOut++ = cNewTerm1;
                if (pOut == pOutMax)
                    break;
            }
            else if (*pIn == ucOldTerm1)
            {
                /*-----
                 * Put 1-char line terminator in place of 2-char terminator.
                 *-----*/
                pIn++;
                *pOut++ = cNewTerm0;
                if (pOut == pOutMax)
                    break;
            }
            else
            {

```

```
        /*-----
        * Treat HALF a line terminator just like a vanilla character
        *-----*/
        *pOut++ = charmap[uIn];
        if (pOut == pOutMax)
            break;
    }
}
else
{
    //---- Do 1-for-1 substitution for vanilla character
    *pOut++ = charmap[uIn];
    if (pOut == pOutMax)
        break;
}
}

*pOutMax = 0;    // null-terminate converted string
}

//----- all done.  return SUCCESS or ILTR_ERR_TRUNC
return rc;
}
```

```

/*-----
* Name:      fldget.c
* Part of:   IntelliLink translation harness (ILTR library)
* Contents:  Functions for getting Field Values from ILIF or TIF
* Functions:
*           ILFldGet      - does Field Mapping to get field value
*           ILFldGetEx    - the guts of ILFldGet, is directly callable
*           FormText
*           GetBaseValue
*           MapFieldValue
*           MapText
*           StripFloatItems
*           ILTRGetField  - gets field value directly, w/o field mapping
*           ILTRGetFieldEx - gets field value directly, w/o field mapping
*
* NOTE:      For all 3 functions, the last argument, "*pLength", is an
*             IN/OUT parameter that is treated quirkily for TEXT fields:
*
* ==> the IN value of *pLength is max size INCLUDING null terminator,
* ==> but the OUT value of *pLength is STRLEN(text) -- not including null.
*
* So to get the value "Fred", you must supply *pLength >= 5,
* and when the call completes you'll have *pLength == 4.
*
* Author:    Copyright (c) IntelliLink, 1992-1995
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*         a "base" DLL rather than being statically linked into every
*         translator. Please ensure that all non-static functions in this
*         module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#define __MSC
#include "iltr.h"
#include "ilx.h"
#define TEMP_NUM_SIZE 10

//-----
//----- WARNING: Programmer discretion advised! This source file contains
//----- intricate and subtle code which must fully understood before attempting
//----- any changes. Remember that the field mapping logic must handle cases
//----- involving one-to-many, many-to-one, and many-to-many fields. No
//----- changes should be attempted to this module without a thorough knowledge
//----- of field mapping logic and a full assessment of consequences.
/*
* NOTE:  as of 4/25/95, this module is both called by TIF code, and makes
*         calls to TIF code, when we are using TIF in place of ILIF.
*-----*/

//----- Function prototypes for internal functions
static int FormText ( ILTR_PTRANSL tr,
                     int nWhich,
                     IL_PSTR lpOutBuf,
                     unsigned int nOutBuf,
                     ILTR_NDX nTarget,
                     ILTR_NDX nSource,
                     IL_PSTR lpField,
                     unsigned int nField );

static long GetBaseValue ( ILTR_PTRANSL tr,
                          int nWhich,
                          char cSrcType,
                          ILTB_ATTRIB lSrcAttribs,
                          IL_PSTR lpBaseName,
                          char cBaseType,
                          ILTB_ATTRIB lBaseAttribs,
                          int nTarAssoc,
                          BOOL16 *pbNegativeBase);

static int MapFieldValue ( ILTR_PTRANSL tr,
                          int nWhich,
                          ILTR_NDX nTarget,
                          ILTR_NDX nSource,

```

```

        IL_PSTR lpOutBuf,
        unsigned int IL_DIST * nOutBuf );

static int MapText ( IL_PSTR psText,
                    unsigned int nMax,
                    unsigned int IL_DIST *pLen,
                    IL_PSTR pszTerm,
                    IL_PSTR charmap,
                    ILUT_PBUFFER pTmpBuf );

static void StripFloatItems ( ILTR_FLDPTR lpFld,
                             IL_PSTR lpBuffer,
                             int nSrcNdx,
                             int nCurNdx );

/*-----
* Name:      ILFldGet
* Purpose: Retrieve and map requested data field
* Input:     Pointer to field name, contents, and max length
* Return:    SUCCESS or error code
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992, 1993
*-----*/
ILBASEFN_int ILFldGet ( ILTR_PTRANSI tr,
                      IL_PSTR lpName,
                      IL_PSTR lpBuffer,
                      unsigned int IL_DIST *pBufLen )
{
    unsigned int bufSize = *pBufLen;
    char szTagDigits[12];
    char szTypeDesc[ILTR_MAX_TYPEDESC];
    char fldtype;
    //??? dam 1/16  UINT32 maxlen;
    INT32 maxlen;
    LONG lSize;
    ILTB_ATTRIB attribs;
    int rc, rc2;

    rc = ILFldGetEx ( tr, lpName, TIF_AUTO,
                     TRUE, // do character mapping if field is non-binary
                     lpBuffer, pBufLen);

    switch (rc)
    {
        case SUCCESS:
        case ILTR_ERR_NODATA:
        case ILTR_ERR_TRUNC:    break;

        default: return rc;
    }

    /*-----
    * Now we may want to add an SST tag to the data, if all signals are GO.
    *-----*/

    //----- If running under a pre-SST stone age app, do nothing
    if (ILTR_VERSION_IS_PRIOR_TO(21))
        return rc;

    //----- If we're NOT importing into a MAIN section, do nothing
    if (ILTR_TargetSST != ILX_SUBSECT_MAIN)
        return rc;

    //----- If TAGGING is disabled, do nothing
    if (ILTR_Flags & ILTR_DISABLE_SST_TAGGING)
        return rc;

    //----- Get field attributes
    rc2 = ILFldTypeEx (tr, lpName, &fldtype, &attribs, &maxlen, szTypeDesc);
    if (rc2 != SUCCESS)
        return ILERROR_S(lpName, ILTR_ERR_INTERNAL_ERROR);

    //-- If field isn't TAGGED, do nothing
    if ((attribs & ILTB_ATT_TAGGED) == 0)
        return rc;

```

```

//----- Get the current _subType field value
lSize = (LONG) sizeof(szTagDigits);
rc2 = ILTRGetFieldEx (tr, ILTR_SUB_TYPE, TIF_AUTO, szTagDigits, &lSize);
if (rc2 != SUCCESS)
    return ILERROR (rc2, ILTR_ERR_INTERNAL_ERROR);

//----- If subtype is zero (ILX_SUBSECT_MAIN) do nothing
if (IL_STRINGS_EQUAL(szTagDigits, "0"))
    return rc;

rc2 = ILSST_AddTag2 ( tr, bufSize, pBufLen, lpBuffer,
                    szTagDigits, szTypeDesc );
if (rc2 == SUCCESS)
{
    if (rc == ILTR_ERR_TRUNC)
        return ILTR_ERR_TRUNC;

    else if (*pBufLen == 0)    // decorated value is ZERO-LENGTH
        return ILTR_ERR_NODATA;

    else
        return SUCCESS;
}
else
    return rc2;    // ILTR_ERR_TRUNC or abnormal error
} //---- ILFldGet

/*-----
* Name:      ILFldGetEx
* Purpose:   Special entrypoint for TIF to call
*            to Retrieve and map requested data field
*
*            Allows control over which value (ORIGINAL, CURRENT, AUTO) you get.
*            Also allows control over character mapping.
*
* Author:    David Boothby, Copyright (c) IntelliLink, 1995
*-----*/
ILBASEFN_int ILFldGetEx ( ILTR_PTRANS� tr,
                        IL_PSTR lpName,
                        int nWhich,
                        BOOLEAN bMapCharsIfNonBinary,
                        IL_PSTR lpBuffer,
                        unsigned int IL_DIST *pBufLen )
{
    int rc;                                // Return code
    unsigned int nLine = 0;                // Line length
    unsigned int nMaxLen;                  // Maximum field length
    unsigned int nTextLen = 0;             // Text field length
    char cTerm[ILTR_MAX_TERM];            // Field delimiter
    char szLineTerm[ILTR_MAX_TERM];        // Line terminator character
    BOOLEAN bFloat = FALSE;               // Is this a floating field?
    BOOLEAN bFoundOne = FALSE;            // Did we find field name?
    BOOLEAN bGotData = FALSE;             // Did we find data in field?
    BOOLEAN bIsMapped = FALSE;            // Is field mapped?
    ILTR_FLDPTR lpFld;                   // Pointer to fields
    ILTR_NDX ndx;                        // Array index

//#ifdef DEBUG
//    char szBuf[100];
//    IL_PRINTF(szBuf, "ILFldGet(%s)\r\n", lpName);
//    OutputDebugString(szBuf);
//#endif

/*----- PROLOGUE -----
* Traditionally, this function was used strictly on IMPORT to
* retrieve and map the contents of a given input field. As of 4/25/95,
* this function is also used during EXPORT, when we are using TIF in place
* of ILIF, and need to have Field Mapping take place.
* (The TIF file uses one app's Field List; so we EXPORT from that APP w/o
* doing any field mapping, but when exporting from the other APP, we have
* to do field mapping.)
*-----

```

```

* The caller supplies the name of the target field (corresponding
* to selected application) and the text of the requested
* field is returned following field mapping. Nevermind the
* references to the IL_SYNC_MEM macro unless running under
* Lotus System Manager.
*-----*/

if (ILTR_phase == ILTR_PHASE30)
{
    /*-----
    * Phase30, which occurs for ILX_V4 TIF-based translation only,
    * is where the Target Translator sanitizes the Source Records,
    * invokes TIF Conflict Resolution, and finally unloads to the
    * Target App. When Target Translator calls ILFldGet, no field
    * mapping should be done, cuz TIF (unlike ILIF) stores data in
    * Target format.
    *-----*/
    long llen = (long) *pBufLen;
    rc = ILTRGetFieldEx (tr, lpName, nWhich, lpBuffer, &llen);
    *pBufLen = (unsigned int) llen;
    return rc;
}

lpBuffer[0] = ILTR_NULL_CHAR;
nMaxLen = *pBufLen;

/*-----
* Verify that the current buffer size is adequate to handle
* the maximum length. The buffer is dynamically re-allocated
* to fit the maximum possible field size if necessary.
*-----*/
IL_SYNC_MEM (ILTR_field.handle, ILTR_field.buffer);
if (*pBufLen > ILTR_field.width)
{
    /*-----
    * Ensure that required field size does not exceed supported
    * maximum size of record.
    *-----*/
    if (*pBufLen > MAX_IF_ALLOC)
        return ILTR_ERR_NOMEM;

    //----- Reallocate field buffer to hold maximum field size
    ILTR_field.width = *pBufLen;
    IL_REALLOC_MEM ( ILTR_field.width,
                     ILTR_field.handle,
                     ILTR_field.buffer );

    //----- Unable to allocate memory for larger buffer size
    if (ILTR_field.buffer == NULL)
    {
        ILTR_field.width = 0;
        return ILTR_ERR_NOMEM;
    }
}

/*-----
* Treat special case fields.
*-----*/
if ( IL_STRINGS_EQUAL(lpName, ILTR_APP_DATA)
    || IL_STRINGS_EQUAL(lpName, ILTR_SUB_TYPE)
    || IL_STRINGS_EQUAL(lpName, ILTR_REP_BASIC)
    || IL_STRINGS_EQUAL(lpName, ILTR_REP_XDATE) )
{
    long nTempLen = (long) ILTR_field.width;
    rc = ILTRGetFieldEx (tr, lpName, nWhich, ILTR_field.buffer, &nTempLen);
    nTextLen = (unsigned int) nTempLen;

    if ((rc != SUCCESS) && (rc != ILTR_ERR_TRUNC))
    {
        //----- Error - unable to retrieve field
        *pBufLen = 0;
        return rc;
    }

    if (nTextLen == 0)

```

```

{
    //----- Found no field value or null field value
    *pBufLen = 0;
    return ILTR_ERR_NODATA;
}

/*-----
 * Now copy data back into caller's buffer. Do this differently for
 * TEXT vs BINARY fields. The only non-BINARY field is _subType.
 *-----*/
if (IL_STRINGS_EQUAL(lpName, ILTR_SUB_TYPE))
{
    if (nTextLen+1 > *pBufLen)
    {
        //----- truncate text to fit into caller's buffer
        nTextLen = *pBufLen - 1;
        rc = ILTR_ERR_TRUNC;
    }

    IL_STRNCPY (lpBuffer, ILTR_field.buffer, nTextLen);
    lpBuffer[nTextLen] = 0;
}
else
{
    /*-----
     * squeeze binary data into caller's buffer
     *-----*/
    if (nTextLen > *pBufLen)
    {
        nTextLen = *pBufLen;
        rc = ILTR_ERR_TRUNC;
    }

    IL_MEMMOVE (lpBuffer, ILTR_field.buffer, nTextLen);
}

//--- pass back exact length (not including any null terminator)
*pBufLen = nTextLen;
return rc;          // SUCCESS or ILTR_ERR_TRUNC
}

//----- Set initial pointer and index to field map table
IL_SYNC_MEM (ILTR_map.hTarget, ILTR_map.pTarget);
lpFld = ILTR_map.pTarget;
ndx = ILTR_map.TopTarget;

/*-----
 * Scan the field list until we find the field name or we reach
 * the end of list. A field index of -1 indicates end of list.
 *-----*/
rc = FAILURE;
while (ndx != -1)
{
    //----- Compare the internal field names for a match
    if (IL_STRICMP (lpFld[ndx].IntName, lpName) == 0)
    {
        //----- Did we find the first item in field?
        if (!bFoundOne)
        {
            /*-----
             * Is this a multi-line field? If so, we will later replace
             * all occurrences of EOS with the application-specific
             * line terminator.
             *-----*/
            if (lpFld[ndx].Attribs & ILTB_ATT_MULTLINE)
                IL_STRCPY (szLineTerm, ILTR_szLineTerm);

            //----- Replace line terminators with spaces in single-line fields.
            else IL_STRCPY (szLineTerm, ILTR_SPACE_STR);

            //----- Signal that field name was located
            bFoundOne = TRUE;
        }
    }
}

/*-----

```

```

    * Now parse and retrieve the text from corresponding source field,
    * if there is one!
    *-----*/
    if (lpFld[ndx].MapField == ILTR_UNMAPPED_BUT_TAGGED)
    {
        /*-----
        * Special case to ensure that ILFldGetEx for an unmapped tagged
        * field will return ILTR_ERR_NODATA. Then the caller of
        * ILFldGetEx may Add a Tag to the field value.
        *-----*/
        *pBufLen = 0;
        return ILTR_ERR_NODATA;
    }

    else if (lpFld[ndx].MapField == ILTR_UNMAPPED)
    {
        /*-----
        * This item isn't mapped, but don't give up quite
        * yet. If we're looking at a multi-item field, we
        * check every item in the multi-item field (note
        * that every one has same Internal Field Name)
        * before giving up. We know it's time to give up
        * when we reach either a field with a different
        * internal field name, or end of list.
        * For now we say "rc=FAILURE", but that setting isn't
        * as conclusive as it sounds!!
        *-----*/
        rc = FAILURE;
    }
    else
    {
        /*-----
        * Indicate that at least one item is mapped and proceed
        * to retrieve the value from the source field.
        *-----*/
        bIsMapped = TRUE;
        rc = MapFieldValue ( tr, nWhich, ndx,
                           lpFld[ndx].MapField, lpBuffer, pBufLen );
    }

    //----- ERROR - out of memory condition
    if (rc == ILTR_ERR_NOMEM )
        return rc;

    //----- Buffer size too small to hold full source value
    if (rc == ILTR_ERR_TRUNC)
    {
        /*-----
        * If flag says we should,
        * map each character to its new value and replace line
        * terminator characters with the application-specific line
        * terminator. Recompute length (for case szLineTerm > 1).
        *-----*/
        if (bMapCharsIfNonBinary)
        {
            int rc2 = MapText ( lpBuffer, nMaxLen, pBufLen, szLineTerm,
                               ILTR_sImportCharMap.buffer, ILTR_pTmpBuf );
            if (rc2 != SUCCESS)
                //---- report error; more serious than ILTR_ERR_TRUNC
                rc = rc2;
        }
        return rc;
    }

    //----- Did we get back a value from source field?
    if (rc == SUCCESS)
    {
        bGotData = TRUE;
        nTextLen = nLine = IL_STRLEN (lpBuffer);
    }

    //----- Re-synchronize pointers in case we reallocated memory
    IL_SYNC_MEM (ILTR_map.hTarget, ILTR_map.pTarget);
    lpFld = ILTR_map.pTarget;

    /*-----

```



```

    * Is the delimiter an end-of-line (represented as ILTR_EOS_CHAR)?
    * If so, truncate any excess field delimiters. This is done
    * to prevent trailing white space between the last field
    * character and the end of line.
    *-----*/
    if (lpFld[ndx].Term == ILTR_EOS_CHAR)
        lpBuffer[nLine] = ILTR_NULL_CHAR;

    /*-----
    * Is this a floating item with no data? If so, avoid
    * placing the delimiter character(s) in the output buffer.
    *-----*/
    bFloat = (BOOLEAN) (lpFld[ndx].Attribs & ILTB_ATT_FLOAT);
    if (bFloat && rc != SUCCESS) ;

    /*-----
    * Append delimiter to text and place a SPACE following
    * comma delimiter characters. Also convert 'x' separator
    * to ' x' for phone extensions.
    *-----*/
    else
    {
        cTerm[0] = (char) lpFld[ndx].Term;
        cTerm[1] = ILTR_NULL_CHAR;
        if (lpFld[ndx].Term == ILX_TERM_COMMA)
            IL_STRCAT (cTerm, ILTR_SPACE_STR);
        if (lpFld[ndx].Term == ILX_TERM_X)
        {
            cTerm[0] = ILTR_SPACE_CHAR;
            cTerm[1] = (char) lpFld[ndx].Term;
            cTerm[2] = ILTR_NULL_CHAR;
        }
        if (nTextLen + IL_STRLEN (cTerm) >= *pBufLen)
            return ILTR_ERR_TRUNC;
        IL_STRCAT (lpBuffer, cTerm);
    }

    /*-----
    * Recalculate the line length if the terminator is an EOS.
    * This is done in case we eliminated white space before the
    * end of line.
    *-----*/
    if (lpFld[ndx].Term == ILTR_EOS_CHAR)
        nLine = IL_STRLEN (lpBuffer);
}

else if (bFoundOne)
    /*-----
    * All done mapping! We have found and processed one
    * or more field table entries that have the same
    * internal name as the requested field, but now we've
    * advanced beyond the last such entry, so it's time
    * to get out of this loop.
    *-----*/
    break;

    /*-----
    * Stop here if the COMBINED field has been successfully processed.
    * This means that a COMBINED field was mapped and had a value.
    * This logic enforces the precedence rule that COMBINED fields
    * have priority over component fields if mapped.
    *-----*/
    if (rc == SUCCESS && (lpFld[ndx].Attribs & ILTB_ATT_COMBINED))
        break;

    //----- Now reset index to next field descriptor
    ndx = lpFld[ndx].NextField;
}

/*-----
* Truncate the field after the last text value to remove any
* trailing delimiter characters.
*-----*/
lpBuffer[nTextLen] = ILTR_NULL_CHAR;

```

```

/*-----
 * Return pointer to field contents. Determine if an error code
 * also needs to be returned. We detect if the field was not found
 * in the list, the field was not mapped, or if the source field
 * contained no data.
 *-----*/
*pBufLen = nTextLen;
if (bFoundOne == FALSE)
    return ILTR_ERR_NOFLD;
if (bIsMapped == FALSE)
    return ILTR_ERR_NOTMAPPED;
if (bGotData == FALSE)
    return ILTR_ERR_NODATA;

/*-----
 * If flag says we should,
 * map each character to its new value and replace line
 * terminator characters with the application-specific line
 * terminator. Recompute length (for case szLineTerm > 1).
 *-----*/
if (bMapCharsIfNonBinary)
{
    rc = MapText ( lpBuffer, nMaxLen, pBufLen, szLineTerm,
                  ILTR_sImportCharMap.buffer, ILTR_pTmpBuf );
    return rc;
}
else
    return SUCCESS;
} //---- ILFldGetEx

/*-----
 * Name:      FormText
 * Purpose:   Format target field item and append to output buffer
 * Input:     Translation structure, output buffer, length of output buffer,
 *            index of target field, index of source field, pointer
 *            to field buffer, length of field buffer
 * Return:    SUCCESS or FAILURE
 * Author:    Mike Blanchette, Copyright (c) IntelliLink, 1993
 *-----*/
static int FormText ( ILTR_PTRANSL tr,
                     int nWhich,
                     IL_PSTR lpOutBuf,
                     unsigned int nOutBuf,
                     ILTR_NDX nTarget,
                     ILTR_NDX nSource,
                     IL_PSTR lpField,
                     unsigned int nField )
{
    int rc = SUCCESS; // Return code
    long nFromBase = 0; // "From" base value
    long nToBase = 0; // "To" base value
    unsigned int nPhoneLen; // Length of phone field
    unsigned int nTotLen; // Total buffer length
    char cItemType; // Item type
    char szArea[ILTR_MAX_PHONE]; // Area code
    char szCountry[ILTR_MAX_PHONE]; // Country code
    char szDefault[ILTR_MAX_PHONE]; // Default phone
    char szExt[ILTR_MAX_PHONE]; // Extension
    char szLabel[ILTR_MAX_PREFIX+2]; // Phone label and delimiters
    char szNumber[ILTR_MAX_PHONE]; // Number
    char szPhone[ILTR_MAX_PHONE]; // Full phone number
    char szTypeDesc[ILTR_MAX_TYPEDESC]; // Phone type description
    ILTR_FLDPTR lpSrc; // Pointer to current source item
    ILTR_FLDPTR lpTar; // Pointer to current target item
    ILTR_NDX nBaseNdx; // Index of "base" field
    BOOL16 bNegativeFrom; // Is From base negative?
    BOOL16 bNegativeTo; // Is To base negative?

    //----- Nothing in field
    if (!nField)
        return ILTR_ERR_NODATA;

    //----- Get pointers to source and target field lists

```

```

IL_SYNC_MEM (ILTR_map.hSource, ILTR_map.pSource);
IL_SYNC_MEM (ILTR_map.hTarget, ILTR_map.pTarget);
lpSrc = ILTR_map.pSource;
lpTar = ILTR_map.pTarget;

/*-----
 * Calculate current output buffer length. Note that data may
 * already exist in the buffer when this routine is called. In
 * the case of multi-item fields, the preceding items will have
 * been formatted and placed in the output buffer. This routine
 * appends to the end of the existing buffer and assumes that
 * the caller has cleared the buffer between fields.
 *-----*/
nTotLen = IL_STRLEN (lpOutBuf);

//----- Is this a phone number field?
if (lpTar[nTarget].Type == ILX_TYPE_PHONE)
{
    /*-----
     * Copy incoming phone number into temporary buffer.
     * Make sure that length does not exceed allowable phone
     * number length.
     *-----*/
    szPhone[0] = '\0';
    nPhoneLen = (nField >= ILTR_MAX_PHONE) ? ILTR_MAX_PHONE-1 : nField;
    IL_STRNCAT (szPhone, lpField, nPhoneLen);

    //----- Note that phone field has been truncated
    if (nPhoneLen != nField)
        ILAddFieldError (tr, lpTar[nTarget].IntName, ILTR_ERR_TRUNC);

    //----- Separate phone number into its constituent parts
    ILSplitPhone ( szPhone,
                   szLabel,
                   szTypeDesc,
                   szCountry,
                   szArea,
                   szNumber,
                   szExt,
                   szDefault );

    /*-----
     * Does phone number need to be stripped of extraneous components?
     * When the KEEPHONE attribute is set on the target item, this
     * means that all phone number components should remain intact
     * in the returned value. If the attribute is not set, then
     * the phone number is constructed strictly using the country
     * code, area code, number, and extension. All other parts are
     * dropped before appending the phone number to output buffer.
     *-----*/
    if (!(lpTar[nTarget].Attribs & ILTB_ATT_KEEPHONE))
    {
        //----- Now reconstruct the phone field without extra parts
        ILMakePhone ( szPhone,
                     NULL,
                     NULL,
                     szCountry,
                     szArea,
                     szNumber,
                     szExt,
                     NULL );

        /*-----
         * Prepend the floating label to phone field. This handles
         * the case where the phone number is a simple floating field
         * such as is the case for the HP 95LX (like "+f-" prefix).
         * Note that this only occurs if the KEEPHONE attribute is
         * turned off since the Label field would otherwise be used
         * to contain the phone label (ex. "<Business>") rather than
         * the floating field prefix. If this is confusing, it may
         * be due to the fact that the Label field is overloaded and
         * used for two distinct purposes (unfortunate but true).
         *-----*/
        if (lpTar[nTarget].Attribs & ILTB_ATT_FLOAT)
        {

```

```

        IL_STRCPY (szNumber, lpTar[nTarget].Label);
        IL_STRNCAT ( szNumber,
                    szPhone,
                    ((ILTR_MAX_PHONE - IL_STRLEN (szNumber)) - 1) );
        IL_STRCPY (szPhone, szNumber);
    }
}

//----- Keep all phone number components
else
{
    /*-----
    * Funny business going on here. Check if a phone label exists
    * in the source field. If not, then first try to use the
    * phone number type as the label if one exists. If this
    * fails, then use the default phone number label contained
    * in the field descriptor. Confused? You should be.
    * This logic exists to address unique problems associated
    * with exporting and re-importing phone number labels
    * between diverse applications like PackRat and Intel AB.
    *-----*/
    if (!szLabel[0] && lpTar[nTarget].Label[0])
    {
        if (szTypeDesc[0])
            IL_STRNCAT (szLabel, szTypeDesc, ILTR_MAX_PREFIX);
        else IL_STRCPY (szLabel, lpTar[nTarget].Label);
    }

    /*-----
    * Does a phone type exist in the source field? If not,
    * apply the default phone type value contained in the field
    * descriptor.
    *-----*/
    if (!szTypeDesc[0] && lpTar[nTarget].TypeDesc[0])
        IL_STRCPY (szTypeDesc, lpTar[nTarget].TypeDesc);

    //----- Rebuild the phone number field in all its glory
    ILMakePhone ( szPhone,
                  szLabel,
                  szTypeDesc,
                  szCountry,
                  szArea,
                  szNumber,
                  szExt,
                  szDefault );
}

//----- Move the phone number to target buffer
//---- (i.e. do type conversion and append result to lpOutBuf)
rc = ILConvertType ( &ILTR_DtTmFmt,
                    lpSrc[nSource].Type,
                    lpSrc[nSource].Attribs,
                    szPhone,
                    IL_STRLEN (szPhone),
                    nFromBase,
                    (nFromBase < 0),
                    lpTar[nTarget].Type,
                    lpTar[nTarget].Attribs,
                    lpOutBuf,
                    nOutBuf,
                    nToBase,
                    (nToBase < 0));

//----- Place note of invalid field in log file
if ( (rc == ILTR_ERR_CANT_CONVERT_TYPE)
    || (rc == ILTR_ERR_CANT_CONVERT_VALUE) )
{
    ILAddFieldError (tr, lpTar[nTarget].IntName, rc);
    //--- pass back "traditional" return code rather than new
    //--- codes which may upset callers of ILFldGet
    rc = ILTR_ERR_CORRUPT_DATA;
}
}

//----- Thank God this is not a phone number

```

```

else
{
    /*-----
    * Place the item prefix in the field (if one exists) after
    * verifying the field length.
    *-----*/
    if (lpTar[nTarget].Label[0])
    {
        IL_STRCPY (szLabel, lpTar[nTarget].Label);
        if ((nTotLen += IL_STRLEN (szLabel)) >= nOutBuf)
            return ILTR_ERR_TRUNC;
        IL_STRCAT (lpOutBuf, szLabel);
    }

    /*-----
    * Does the source field of number type have a dependency?
    * A number field may need special processing if it includes
    * a dependency rule that associates it either with a date
    * or time field. This mechanism enables number fields to
    * represents relative date and relative time values based
    * on some other date and time field in the record.
    *-----*/
    if (lpSrc[nSource].Type == ILX_TYPE_NUMBER && lpSrc[nSource].Assoc)
    {
        /*-----
        * Is the number field mapped to a date or time field? If so,
        * we need to convert it to an actual date or time field.
        *-----*/
        cItemType = lpTar[nTarget].Type;
        if (cItemType == ILX_TYPE_DATE || cItemType == ILX_TYPE_TIME)
        {
            /*-----
            * Determine name of dependent field in intermediate record.
            * Return no value if dependent field is unmapped.
            *-----*/
            nBaseNdx = (abs (lpSrc[nSource].Assoc)) - 1;
            if (lpSrc[nBaseNdx].MapField == ILTR_UNMAPPED)
                return SUCCESS;

            /*-----
            * Retrieve the value of base field. The "base" field is
            * simply the field that will be used to compute the actual
            * date or time value.
            *-----*/
            nFromBase = GetBaseValue ( tr, nWhich,
                                      lpTar[nTarget].Type,
                                      lpTar[nTarget].Attribs,
                                      lpSrc[nBaseNdx].IntName,
                                      lpSrc[nBaseNdx].Type,
                                      lpSrc[nBaseNdx].Attribs,
                                      lpSrc[nSource].Assoc,
                                      &bNegativeFrom);
        }
    }

    /*-----
    * Does the target date or time field have a dependency? The
    * inverse of the logic described for source fields applies here.
    * A date or time field may have a dependency on a number field.
    * This means that it needs to be computed based on the number
    * field before being returned.
    *-----*/
    cItemType = lpSrc[nSource].Type;
    if (cItemType == ILX_TYPE_DATE || cItemType == ILX_TYPE_TIME)
    {
        /*----- Is the source field have a dependency on a number?
        if (lpTar[nTarget].Type == ILX_TYPE_NUMBER && lpTar[nTarget].Assoc)
        {
            /*-----
            * Get name of dependent field.
            * Return no value if dependent field is unmapped.
            *-----*/
            nBaseNdx = (abs (lpTar[nTarget].Assoc)) - 1;
            nBaseNdx = lpTar[nBaseNdx].MapField;
            if (nBaseNdx == ILTR_UNMAPPED)

```

```

        return SUCCESS;

/*-----
 * Get value of dependent field from intermediate file.
 * The value returned contains a positive or negative
 * number representing a relative date or relative time.
 * This is a long value encoded using either IL_DateEncoe
 * or IL_TimeEncode functions. The base value will be
 * used in a later call to the function ILConvertType.
 *-----*/
nToBase = GetBaseValue ( tr, nWhich,
                        lpSrc[nSource].Type,
                        lpSrc[nSource].Attribs,
                        lpSrc[nBaseNdx].IntName,
                        lpSrc[nBaseNdx].Type,
                        lpSrc[nBaseNdx].Attribs,
                        lpTar[nTarget].Assoc,
                        &bNegativeTo);
    }
}

/*-----
 * Convert item to appropriate format and place in output buffer.
 * This call will result in data conversions between field types.
 * It will also handle the computation of relative date and time
 * fields. Data validation of typed fields is also performed
 * as part of this call to ILConvertType.
 *
 * (ILConvertType does type conversion and appends result to lpOutBuf)
 *-----*/
rc = ILConvertType ( &ILTR_DtTmFmt,
                    lpSrc[nSource].Type,
                    lpSrc[nSource].Attribs,
                    lpField,
                    nField,
                    nFromBase,
                    bNegativeFrom,
                    lpTar[nTarget].Type,
                    lpTar[nTarget].Attribs,
                    lpOutBuf,
                    nOutBuf,
                    nToBase,
                    bNegativeTo);

//----- Place note in log file if we couldn't convert field
if ( (rc == ILTR_ERR_CANT_CONVERT_TYPE)
    || (rc == ILTR_ERR_CANT_CONVERT_VALUE) )
{
    ILAddFieldError (tr, lpTar[nTarget].IntName, rc);
    //--- pass back "traditional" return code rather than new
    //--- codes which may upset callers of ILFldGet
    rc = ILTR_ERR_CORRUPT_DATA;
}

//----- All better now
return rc;

} //---- FormText

/*-----
 * Name:      GetBaseValue
 * Purpose:   Retrieve base value from intermediate file
 * Input:     Translation structure, source item type, base field name,
 *            base field type, index of dependent field
 * Return:    Base value or 0L in case of error
 * Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
 *-----*/
static long GetBaseValue ( ILTR_PTRANSL tr,
                          int nWhich,
                          char cSrcType,
                          ILTB_ATTRIB lSrcAttribs,
                          IL_PSTR lpBaseName,
                          char cBaseType,

```

```

        ILTB_ATTRIB lBaseAttribs,
        int nTarAssoc,
        BOOL16 * pbNegativeBase)
{
    int month, day, year;           // Date components
    int hours, mins;               // Time components
    int rc;                        // Return code
    long len;                      // Data length
    long nValue;                   // Base field value
    char szBase[TEMP_NUM_SIZE];    // Temporary base value
    char szNewBase[TEMP_NUM_SIZE]; // Temporary base value

    //----- Clear the return value
    nValue = 0L;

    /*-----
    * Retrieve the value of dependent field from the intermediate file.
    * Note that this assumes that the dependent field is a single-item
    * field, otherwise this scheme does not work. This means that all
    * number, date, and time fields having dependencies MUST be single
    * fields and not be part of a multi-item field. Seems like a fair
    * assumption at this time but one to keep in mind for the future.
    *-----*/
    len = sizeof (szBase);
    rc = ILTRGetFieldEx (tr, lpBaseName, nWhich, szBase, &len);

    //----- Something is definitely wrong here
    if (rc)
        return nValue;

    /*-----
    * Convert base type to source type and place. This call validates
    * the value of the base field and converts types if necessary.
    *-----*/
    IL_MAKE_STRING_NULL(szNewBase);
    //----- do type conversion and append result to szNewBase
    if (ILConvertType ( &ILTR_DtTmFmt,
                        cBaseType,
                        lBaseAttribs,
                        szBase,
                        IL_STRLEN (szBase),
                        0L,
                        FALSE,
                        cSrcType,
                        lSrcAttribs,
                        szNewBase,
                        sizeof (szNewBase),
                        0L,
                        FALSE))
        return nValue;

    //----- Convert source date to relative date field
    if (cSrcType == ILX_TYPE_DATE)
    {
        IL_AlphaToDate (szNewBase, &month, &day, &year);
        IL_DateEncode (month, day, year, &nValue);
    }

    //----- Convert source time field to relative time field
    else if (cSrcType == ILX_TYPE_TIME)
    {
        IL_AlphaToTime (szNewBase, &hours, &mins);
        IL_TimeEncode (hours, mins, 0, &nValue);
    }

    //----- Invalid type for field association
    else return nValue;

    /*-----
    * Adjust the base value to reflect sign direction. If the dependent
    * field is negative, it means that the base value will be deducted
    * from the source value. Otherwise, the base value will be added.
    *-----*/
    if (nTarAssoc < 0)
    {

```



```

        nValue = 0 - nValue;
        *pbNegativeBase = TRUE;
    }
    else
        *pbNegativeBase = FALSE;

    //----- Return without error
    return nValue;
} //----- GetBaseValue

/*-----
* Name:      MapFieldValue
* Purpose:   Map and retrieve the value of associated source field
* Input:     Translation structure, index of target field, index of
*            source field, pointer to field buffer, buffer length
* Return:    SUCCESS or FAILURE
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/
static int MapFieldValue ( ILTR_PTRANSL tr,
                           int nWhich,
                           ILTR_NDX nTarget,
                           ILTR_NDX nSource,
                           IL_PSTR lpOutBuf,
                           unsigned int IL_DIST * nOutBuf )
{
    int nInitialRC;                // Remember TRUNCATE error
    int rc;                        // Return code
    long nTempLen;                 // Temporary length field
    unsigned int nFldLen;          // Actual field length
    BOOLEAN bStrip;               // Must the prefix be stripped?
    IL_PSTR lpNext;                // Pointer to next field item
    IL_PSTR lpStart;              // Pointer to start of field item
    ILTR_FLDPTR lpFld;             // Pointer to current field item
    ILTR_NDX nCurNdx;             // Index to current field item
    ILTR_NDX nFirstNdx;           // Index to first field item

    //##### DEBUG
    //  char szBuf[100];
    //  IL_PRINTF(szBuf, "  MapFieldValue(%d,%d)\r\n", nSource, nTarget);
    //  OutputDebugString(szBuf);
    //#####

    //----- Set initial pointer and index values
    IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);
    IL_SYNC_MEM (ILTR_map.hSource, ILTR_map.pSource);
    lpFld = ILTR_map.pSource;
    nInitialRC = SUCCESS;

    //----- Retrieve the field from intermediate file
    nTempLen = (long) ILTR_field.width;
    rc = ILTRGetFieldEx ( tr, lpFld[nSource].IntName, nWhich,
                        ILTR_field.buffer, &nTempLen );

    /*-----
    * Warning - field truncated on input.  Field buffer too small
    * for incoming field.  Proceed with formatting the field anyway.
    *-----*/
    if (rc == ILTR_ERR_TRUNC)
    {
        if (*nOutBuf > (unsigned int) nTempLen)
            *nOutBuf = (unsigned int) nTempLen;
        ILTR_field.buffer[ILTR_field.width - 1] = '\0';
        nInitialRC = ILTR_ERR_TRUNC;
        rc = SUCCESS;
    }

    //----- Oops!  Something is wrong here.
    if (rc != SUCCESS)
        return rc;

    //----- Seems that there is nothing in the field
    if (nTempLen == 0)
        return ILTR_ERR_NODATA;

```

```

/*-----
 * Is this a floating item? If so, locate it directly in the field
 * buffer using the item prefix and leave.
 *-----*/
if ((lpFld[nSource].Attribs & ILTB_ATT_FLOAT))
{
    /*-----
     * Keep the prefix prepended to the field?
     * The prefix is only retained for floating phone number items.
     *-----*/
    if (lpFld[nSource].Type == ILX_TYPE_PHONE)
        bStrip = FALSE;
    else bStrip = TRUE;

    /*-----
     * Retrieve the floating item from the field buffer and
     * return formatted field.
     *-----*/
    rc = ILFldFloat ( ILTR_field.buffer,
                     &lpStart,
                     &nFldLen,
                     &lpFld[nSource],
                     bStrip );

    //----- Did we find the floating field?
    if (rc)
    {
        //----- Now format the output string and leave
        if ((rc = FormText ( tr, nWhich,
                           lpOutBuf,
                           *nOutBuf,
                           nTarget,
                           nSource,
                           lpStart,
                           nFldLen )))
            return rc;
        else return nInitialRC;
    }

    //----- Floating field not present in buffer
    else return ILTR_ERR_NODATA;
}

/*-----
 * Is this the first item for given source field?
 * Walk the field map table in reverse order until the first
 * item is located.
 *-----*/
nFirstNdx = nSource;
while (lpFld[nFirstNdx].ItemNo != 1)
    nFirstNdx = lpFld[nFirstNdx].PriorField;

/*-----
 * Is the first item in the field a COMBINED item?
 * Skip the COMBINED item if request is for a component item.
 *-----*/
if (lpFld[nFirstNdx].Attribs & ILTB_ATT_COMBINED)
{
    if (nFirstNdx != nSource)
    {
        nFirstNdx++;
        StripFloatItems (lpFld, ILTR_field.buffer, nSource, nFirstNdx);
    }
    /*-----
     * But don't strip floating items if the combined field itself is mapped,
     * as opposed to having one of the items within a combined field be mapped.
     *-----*/
}
else
{
    /*-----
     * Not a floating item. Remove all occurrences of floating items
     * from the field buffer before proceeding. This will leave
     * the buffer containing only non-floating items in their

```

```

    * ordinal positions.
    *-----*/
    StripFloatItems (lpFld, ILTR_field.buffer, nSource, nFirstNdx);
}

/*-----
 * Prepare to walk the linked list of field items.
 * Start by retrieving first item from field buffer.
 *-----*/
nCurNdx = nFirstNdx;
lpStart = lpNext = ILTR_field.buffer;
nFldLen = ILFldItem ((IL_PSTR IL_DIST *) &lpNext, lpFld, nCurNdx);

//----- Parse all source fields until the desired field is located
while (nCurNdx != -1)
{
    /*-----
     * Make sure that the field contains data and that it
     * corresponds to the desired field name.
     *-----*/
    if (IL_STRICMP (lpFld[nCurNdx].IntName, lpFld[nSource].IntName) != 0)
        break;

    /*-----
     * Skip all floating item nodes since all occurrences of floating
     * items have been removed from the buffer by this time.
     *-----*/
    if ((lpFld[nCurNdx].Attribs & ILTB_ATT_FLOAT))
    {
        nCurNdx = lpFld[nCurNdx].NextField;
        continue;
    }

    //----- Found the item we want
    if (nCurNdx == nSource)
    {
        //----- Unfortunately no value in the item
        if (nFldLen == 0)
            break;
        else
        {
            //----- Format the output string
            if ((rc = FormText (tr, nWhich,
                               lpOutBuf,
                               *nOutBuf,
                               nTarget,
                               nSource,
                               lpStart,
                               nFldLen)))
                return rc;
            else return nInitialRC;
        }
    }

    //----- Go to the next item node and get next item value from field
    nCurNdx = lpFld[nCurNdx].NextField;
    lpStart = lpNext;
    if (nCurNdx != -1)
        nFldLen = ILFldItem ((IL_PSTR IL_DIST *) &lpNext, lpFld, nCurNdx);
}

//----- Return without a field value
return ILTR_ERR_NODATA;
} //---- MapFieldValue

/*-----
 * Name:      MapText
 * Purpose: Replace EOS with application-specific line terminator
 * Input:   Pointer to buffer, buffer length, pointer to field terminator
 * Return:  SUCCESS or error code
 * Author:  Roger Duchesneau, Copyright (c) IntelliLink, 1994
 *-----*/
static int MapText ( IL_PSTR psText,

```

```

        unsigned int uiMax,
        unsigned int IL_DIST *pLen,
        IL_PSTR pszTerm,
        IL_PSTR charmap,
        ILUT_PBUFFER pTmpBuf )
{
    int nRc;
    INT32 lMax = min (ILTR_MAX_FIELDLLENGTH, (INT32) uiMax);

    nRc = IMapChars ( pszText,
        (INT32) *pLen,
        ILTR_EOS_STR,          // old line terminator ("\xFF")
        pszTerm,              // new line terminator (e.g. "\r\n")
        charmap,
        lMax,
        pTmpBuf );

    if (pTmpBuf->pBuffer != NULL)
        IL_SAFE_STRINGCOPYN (pszText, pTmpBuf->pBuffer, (unsigned int) lMax);

    *pLen = IL_STRLEN(pszText);

    return nRc;
} //---- MapText

/*-----
* Name:      StripFloatItems
* Purpose:   Remove all floating items from buffer
* Input:     Pointer to field node, pointer to buffer,
* Return:    SUCCESS or error code
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1993
*-----*/
static void StripFloatItems ( ILTR_FLDPTR lpFld,
                             IL_PSTR lpBuffer,
                             int nSrcNdx,
                             int nCurNdx )
{
    int rc;                      // Return code
    unsigned int nLen;           // Item length
    int nToMove;                // # characters to move
    IL_PSTR lpItem;             // Pointer to item value

    //----- Scan all item nodes for floating items
    while (nCurNdx != -1)
    {
        //----- Make sure that we have not gone beyond relevant field
        if (IL_STRICMP (lpFld[nCurNdx].IntName, lpFld[nSrcNdx].IntName) != 0)
            break;

        //----- Is this a floating item?
        if ((lpFld[nCurNdx].Attribs & ILTB_ATT_FLOAT) == ILTB_ATT_FLOAT)
        {
            //----- Get its position in the string
            rc = ILFldFloat ( lpBuffer,
                             &lpItem,
                             &nLen,
                             &lpFld[nCurNdx],
                             TRUE );

            //----- Now remove it from the string
            if (rc)
            {
                lpItem -= IL_STRLEN (lpFld[nCurNdx].Label);
                nLen += IL_STRLEN (lpFld[nCurNdx].Label) + 1;
                nToMove = (lpBuffer+IL_STRLEN(lpBuffer))-(lpItem+nLen) + 1;

                //----- Slide buffer over, or zap if last item in buffer
                if (nToMove > 0)
                    IL_MEMMOVE ( lpItem,
                                lpItem+nLen,
                                nToMove );
                else
                    lpItem[0] = '\0';
            }
        }
    }
}

```

```

    }
}

//----- Go to the next item node and get next item value from field
nCurNdx = lpFld[nCurNdx].NextField;
}

} //----- StripFloatItems

/*-----
 * ILTRGetField
 * public function to get unmapped field value, either from ILIF or from TIF.
 * when reading from TIF, always uses option 'TIF_AUTO'
 *-----*/
ILBASEFN_int ILTRGetField (
    ILTR_PTRANSI tr,
    IL_PSTR lpszFieldName,
    IL_PANY lpszFieldValue,
    long IL_DIST *pFieldLength )
{
    int rc = ILTRGetFieldEx ( tr, lpszFieldName, TIF_AUTO,
                             lpszFieldValue, pFieldLength );
    return rc;
} //----- ILTRGetField

/*-----
 * ILTRGetFieldEx -- called by ILFldGetEx & ILTRGetField & from repeat.c
 *-----*/
ILBASEFN_int ILTRGetFieldEx
(
    ILTR_PTRANSI tr,
    IL_PSTR lpszFieldName,
    int nWhich,
    IL_PANY lpszFieldValue,
    long IL_DIST *pFieldLength )
{
    int rc;

    /*-----
     * Now we get the field either from ILIF or from TIF.
     * The tr structure member ILTR_phase tells us which to use.
     *-----*/
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        rc = ILIFGetField ( ILTR_view,
                           ILTR_rec.buffer, ILTR_rec.width,
                           lpszFieldName, lpszFieldValue, pFieldLength );

        //----- Map the ILIF error to an ILTR error.
        rc = ILMaPFErrToTRErr(rc);
    }
    else
    {
        /*-----
         * To get from TIF, we call a special TIF entryptoint that copies
         * result into caller's buffer and applies truncation rules just
         * like ILIF does. (Note that standard ILTIFGetField returns a
         * pointer to an un-truncated field value in a buffer owned by
         * TIF, and returns a length that is exact for BINARY
         * fields, and is exactly STRLEN(text)+1 for text values.) But
         * ILTIFGetAndCopy does copying and truncation, and for text
         * fields it returns length=STRLEN(text), not STRLEN(text)+1.
         *-----*/
        rc = ILTR_ILTIFGetAndCopyField ( tr, lpszFieldName, nWhich,
                                         pFieldLength, lpszFieldValue );
    }

    return rc;
} //----- ILTRGetFieldEx

```

```

/*-----
* Module Name:      fldput.c
* Part of:          the IntelliLink Translation Harness (ILTR library)
* Contents:         2 public functions for putting a field to ILIF or TIF:
*
*                   ILFldPut      -- high-level function
*                   - ILTRPutField -- low-level function
*
* ILFldPut is called by every single translator.
* ILTRPutField is called from several ILTR modules.
*
* Copyright (c) IntelliLink, 1992-1995
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*         a "base" DLL rather than being statically linked into every
*         translator. Please ensure that all non-static functions in this
*         module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#include "iltr.h"

#define EXIT_WITH_ERROR(e) {rc=e; goto Exit;}

/*-----
* Name:      ILFldPut
* Purpose:   High-level function to output a field to ILIF or TIF
*            after doing character mapping.
* Input:     Pointer to field name and contents
*
* Note:      Pass exact bytecount of field value as "len" arg. For
*            non-binary fields do NOT include the null terminator
*            byte in the bytecount. For "Job" pass len=3.
*
*            The input value need not be null-terminated. Passing
*            text="JobCode" with len=3 is equivalent to passing
*            text="Job" with len=3.
*
* Return:    SUCCESS, or error code
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/
ILBASEFN_int ILFldPut ( ILTR_PTRANSI tr,
                        IL_PSTR name,
                        IL_PSTR text,
                        unsigned int len )
{
    int rc;
    BOOLEAN isaTextField;
    BOOLEAN bMustMapChars;
    BOOLEAN bSupplyDataToFiltersMechanism;
    BOOLEAN bSpecialFanningAdjustment;

    char szTypeDesc[ILTR_MAX_TYPEDESC];
    char fldtype = ILX_TYPE_TEXT;
    //??? dam 1/16  UINT32 maxlen = 0;
    INT32 maxlen = 0;
    ILTB_ATTRIB attribs = 0;

    //----- Get field attributes
    rc = ILFldTypeEx (tr, name, &fldtype, &attribs, &maxlen, szTypeDesc);

    //----- If failure, field not mapped - all done
    if (rc != SUCCESS)
        return ILTR_ERR_NOFLD;

    isaTextField = (fldtype != ILX_TYPE_BINARY);

    //----- If this is a TAGGED field, strip tag and put it in _subType field
    //----- (but don't do this when running under a pre-SST stone age App)
    if ((attribs & ILTB_ATT_TAGGED) && ILTR_VERSION_IS_AT_LEAST(21))
    {
        char szTag[ILTR_MAX_TAG_LEN];
        rc = ILSST_StripTag (tr, &text, &len, szTypeDesc, szTag);
        if (rc != SUCCESS)

```

```

        return rc;

    if (IL_STRING_ISNT_NULL(szTag))
    {
        rc = ILTRPutField ( tr, ILTR_SUB_TYPE,
                           szTag, IL_STRLEN(szTag),
                           FALSE, FALSE, FALSE );
        if (rc != SUCCESS)
            return ILERROR (rc, ILTR_ERR_INTERNAL_ERROR);
    }
}

/*-----
 * If field type is non-binary text, do character mapping and pass
 * field data on to the FILTERS mechanism. Then, for ALL field types,
 * put field into the intermediate file (either ILIF or ILTIF).
 *-----*/
bMustMapChars = isaTextField;
bSupplyDataToFiltersMechanism = isaTextField;
bSpecialFanningAdjustment = FALSE;

rc = ILTRPutField ( tr, name, text, (INT32) len,
                   bSpecialFanningAdjustment,
                   bMustMapChars,
                   bSupplyDataToFiltersMechanism );

return rc;
}

/*-----
 * Name:      ILTRPutField
 * Purpose:   Low-level function to output a field to ILIF or TIF
 *
 * 'bSpecialFanningAdjustment' is TRUE when REPEAT.C UpdateDateField
 * function calls this function.
 *
 * NOTE:      this function may or may not do character mapping, and may or
 * may not pass data on to the FILTERS mechanism, depending on the
 * last two boolean arguments that it takes.
 *
 * Of course for environments that don't support FILTERS, nothing
 * is passed on to the FILTERS mechanism.
 *
 * The reusable buffer 'ILTR_pTmpBuf' is used here
 *
 * Called from: this module & putrep.c & repeat.c & sst.c
 *-----*/
ILBASEFN_int ILTRPutField ( ILTR_PTRANSI tr,
                           IL_PSTR name,
                           IL_PSTR text,
                           INT32 len,
                           BOOLEAN bSpecialFanningAdjustment,
                           BOOLEAN bDoCharacterMapping,
                           BOOLEAN bSupplyDataToFiltersMechanism )
{
    int rc;
    IL_PSTR pOutBuf;

    if (bDoCharacterMapping)
    {
        /*-----
         * Do character mapping, and put result string in ILTR_pTmpBuf.
         * ILTR_pTmpBuf is managed as a reusable buffer to minimize heap activity
         *-----*/
        rc = ILMMapChars ( text,
                           len,
                           // may or may not equal strlen(text)
                           ILTR_szLineTerm, // old line terminator (e.g. "\r\n")
                           ILTR_EOS_STR,    // new line terminator ("\xFF")
                           ILTR_sExportCharMap.buffer,
                           ILTR_MAX_FIELDLENGTH,
                           ILTR_pTmpBuf );
        if (rc != SUCCESS)
            return ILERROR (rc, ILTR_ERR_INTERNAL_ERROR);

        pOutBuf = ILTR_pTmpBuf->pBuffer;
    }
}

```



```

    len = IL_STRLEN (pOutBuf);
}
else
    pOutBuf = text;

if (bSupplyDataToFiltersMechanism)
{
    #ifdef ILWIN
        /*----- Is there a filter active?
        if (ILTR_nFilterID != -1)
        {
            rc = ILFldParse (tr, name, pOutBuf, (int) len);
            if (rc != SUCCESS)
                EXIT_WITH_ERROR (ILERROR(rc, ILTR_ERR_INTERNAL_ERROR));
        }
        #else
            ; // NO-OP
        #endif
    }

    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
    {
        /*-----
        * Using TIF as intermediate file. Put field to TIF. We've already
        * done character mapping, so tell TIF not to. We pass "len" to TIF,
        * but BEWARE: TIF ignores "len" for non-binary fields; it gets STRLEN
        * of string (works fine iff string is NULL-terminated.
        *-----*/
        rc = ILTR_ILTIFPutFieldEx ( tr, name, (IL_PANY) pOutBuf,
                                   len, // ignored for non-binary fields!!
                                   bSpecialFanningAdjustment,
                                   FALSE );

        if (rc != SUCCESS)
            rc = ILERROR(rc, ILTR_ERR_INTERNAL_ERROR);
    }
    else
    {
        /*----- NOT USING TIF: Place field in ILIF output record.
        IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);

        /*-----
        * Keep trying to add field to ILIF record, expanding record as
        * necessary; until we succeed or give up when maxed out.
        *-----*/
        for (;;)
        {
            rc = ILIFPutField ( ILTR_view,
                               ILTR_rec.buffer,
                               ILTR_rec.width,
                               name,
                               (IL_PANY) pOutBuf,
                               (long) len);

            /*----- break out of loop on SUCCESS or serious error
            if (rc != ILIF_ERR_NOROOM)
                break;

            /*-----
            * Increment buffer size and verify that it does not exceed max
            *-----*/
            ILTR_rec.width += MAX_IF_INCR;
            if (ILTR_rec.width > MAX_IF_ALLOC)
            {
                ILTR_rec.width -= MAX_IF_INCR; // back to currently allocated size
                EXIT_WITH_ERROR (ILTR_ERR_RECORD_TOO_BIG);
            }

            /*----- Reallocate buffer to larger size.
            IL_REALLOC_MEM (ILTR_rec.width, ILTR_rec.handle, ILTR_rec.buffer);
            if (ILTR_rec.buffer == NULL)
                EXIT_WITH_ERROR (ILTR_ERR_NOMEM); // allocation failure
        }
    }
}

```

```
        //----- Map the ILIF error to an ILTR error.  
        rc = ILMapIFErrToTRErr(rc);  
    }  
  
Exit:  
    return rc;  
}
```

```

/*-----
 * File Name:      fldtype.c
 * Part of:        IntelliLink Translation Harness (ILTR) library
 * Entrypoints:    ILFldType and ILFldTypeEx
 *                 Copyright (c) IntelliLink Corp., 1993-1995
 *-----*/

/*-----
 * NOTE:  this is an "ILBASE" module, which means that it may be built into
 *         a "base" DLL rather than being statically linked into every
 *         translator. Please ensure that all non-static functions in this
 *         module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
 *-----*/

#define _MSC
#define ILX_NO_GLOBALS
#include "ilxapi.h"

/*-----
 * GetTypeFromMap -- old-fashioned way to get field type
 *-----*/
static int GetTypeFromMap
( ILTR_PTRANSL tr,
  IL_PSTR label,
  IL_PSTR type,
  ILTB_ATTRIB IL_DIST *attribs,
  INT32 IL_DIST *pFldSize,
  IL_PSTR pszTypeDesc );
// Internal field name to use
// Returned field type
// Returned field attributes
// Returned MAXIMUM field size
// Pointer to string for TypeDesc

/*-----
 * GetTypeFromTable -- new-fangled way to get field type
 *-----*/
static int GetTypeFromTable
( ILTR_PTRANSL tr,
  IL_PSTR label,
  BOOLEAN bInvertedLookup,
  IL_PSTR type,
  ILTB_ATTRIB IL_DIST *attribs,
  INT32 IL_DIST *pFldSize,
  IL_PSTR pszTypeDesc );
// Internal field name to use
// Invert usual lookup rules?
// Returned field type
// Returned field attributes
// Returned MAXIMUM field size
// Pointer to string for TypeDesc

/*-----
 * Name:      ILFldType
 * Purpose:   Obtain the field type given its internal name
 * Input:     Translation structure, field label, field type, attributes
 * Return:    SUCCESS or FAILURE if field name not found
 * Author:    Mike Blanchette, Copyright (c) IntelliLink, 1993
 *-----*/
ILBASEFN_int ILFldType
( ILTR_PTRANSL tr,
  IL_PSTR label,
  IL_PSTR type,
  unsigned long IL_DIST *attribs )
{
  INT32 lSize;
  int rc = ILFldTypeEx ( tr, label, type, attribs, &lSize, NULL );
  return rc;
}

/*-----
 * Name:      ILFldTypeEx
 * Purpose:   Obtain the field type and several more useful bits of info:
 *
 *           Field Attributes, MAX Field Size, and the 'TypeDesc'
 *           string which TIF uses for Default Value.
 *
 *           NOTE:  if last arg is NULL the TypeDesc string is not
 *                 copied back to the caller.
 *
 * Input:     Translation structure, field label, field type, attributes
 * Return:    SUCCESS or FAILURE if field name not found
 * Author:    Mike Blanchette, Copyright (c) IntelliLink, 1993
 *-----*/
ILBASEFN_int ILFldTypeEx
( ILTR_PTRANSL tr,
  IL_PSTR label,
  // Get field type, etc.
  // Pointer to translation record
  // Internal field name to use

```

```

        IL_PSTR type,                // Returned field type
        ILTB_ATTRIB IL_DIST *attrs,  // Returned field attributes
        INT32 IL_DIST *pFldSize,     // Returned MAXIMUM field size
        IL_PSTR pszTypeDesc )        // Pointer to string for TypeDesc
{
    int rc;

    type[0] = '\0'; // return NULL field type if lookup fails

    if (ILTR_VERSION_IS_AT_LEAST(14) && ILTR_pTableInfo != NULL)
        rc = GetTypeFromTable ( tr, label, FALSE, type,
                                attrs, pFldSize, pszTypeDesc );
    else
        rc = GetTypeFromMap (tr, label, type, attrs, pFldSize, pszTypeDesc);

    return rc;
}

/*-----
 * Name:      ILFldTypeInvertedLookup
 * Purpose: Obtain the field type and several more useful bits of info:
 *
 *           Analogous to ILFldTypeEx, but the rules for deciding which
 *           field list to scan are inverted.
 *
 *           Called from ILTIF.CPP\ILTIFGetAndCopyField when handling a
 *           "nested" getfield request. This probably only happens in
 *           Phase40 of synchronization, when we're importing updates
 *           into the "source app" and having to do field mapping.
 *
 * Author: David Boothby, Copyright (c) IntelliLink, 1995
 *-----*/
ILBASEFN_int ILFldTypeInvertedLookup
( ILTR_PTRANSL tr,                // Pointer to translation record
  IL_PSTR label,                 // Internal field name to use
  IL_PSTR type,                 // Returned field type
  ILTB_ATTRIB IL_DIST *attrs,    // Returned field attributes
  INT32 IL_DIST *pFldSize,       // Returned MAXIMUM field size
  IL_PSTR pszTypeDesc )          // Pointer to string for TypeDesc
{
    int rc;

    type[0] = '\0'; // return NULL field type if lookup fails

    if (ILTR_VERSION_IS_AT_LEAST(14) && ILTR_pTableInfo != NULL)
        rc = GetTypeFromTable ( tr, label, TRUE, type,
                                attrs, pFldSize, pszTypeDesc );
    else
        //--- inverted lookup shouldn't happen in such a rustic setting
        return ILERROR(0, ILTR_ERR_INTERNAL_ERROR);

    return rc;
}

/*-----
 * GetTypeFromMap -- old-fashioned way to get field type
 *-----*/
static int GetTypeFromMap
( ILTR_PTRANSL tr,                // Internal field name to use
  IL_PSTR label,                 // Returned field type
  IL_PSTR type,                 // Returned field attributes
  ILTB_ATTRIB IL_DIST *attrs,    // Returned MAXIMUM field size
  INT32 IL_DIST *pFldSize,       // Pointer to string for TypeDesc
  IL_PSTR pszTypeDesc )          // Pointer to string for TypeDesc
{
    int ndx = 0;                // Array position of name
    ILTR_FLDPTR fld;            // Pointer to fields
    ILTR_NDX cur;               // Index to current field

    //---- Are we importing? If so, use the list of TARGET fields.
    if (ILTR_direction == ILTR_IMPORT)
    {
        IL_SYNC_MEM (ILTR_map.hTarget, ILTR_map.pTarget);
        fld = ILTR_map.pTarget;
    }

```

```

    cur = ILTR_map.TopTarget;
}

//----- Are we exporting? If so, use the list of SOURCE fields.
else
{
    IL_SYNC_MEM (ILTR_map.hSource, ILTR_map.pSource);
    fld = ILTR_map.pSource;
    cur = ILTR_map.TopSource;
}

//----- Scan the field list until we find the given field label.
while (cur != -1)
{
    /*-----
    * Does the current field label match the one we need?
    * Stop scanning the list when we find first match.
    *-----*/
    if (IL_STRICMP (fld[cur].IntName, label) == 0)
        break;

    //----- Label does not match. Scan next item in the list.
    cur = fld[cur].NextField;
}

//----- Reached end of list without finding required field label.
if (cur == -1)
    return FAILURE;

//----- Set the attributes and type.
*attribs = fld[cur].Attribs;
*type = fld[cur].Type;
*pFldSize = fld[cur].Width;

if (pszTypeDesc != NULL)
    IL_SAFE_STRINGCOPYN (pszTypeDesc, fld[cur].TypeDesc, ILTR_MAX_TYPEDESC);

//----- Return without error.
return SUCCESS;
}

/*-----
* GetTypeFromTable -- new-fangled way to get field type
*-----*/
static int GetTypeFromTable
(
    ILTR_PTRANSL tr,
    IL_PSTR label,
    BOOLEAN bInvertedLookup,
    IL_PSTR type,
    ILTB_ATTRIB IL_DIST *attribs,
    INT32 IL_DIST *pFldSize,
    IL_PSTR pszTypeDesc )
// Internal field name to use
// Invert usual lookup rules?
// Returned field type
// Returned field attributes
// Returned MAXIMUM field size
// Pointer to string for TypeDesc
{
    ILTR_PFLDMAP pMap = &ILTR_pTableInfo->sFieldMap;
    ILTR_FLDPTR pList;
    int count;
    int i;

    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        if (bInvertedLookup)
            //--- inverted lookup shouldn't happen in such a rustic setting
            return ILERROR(0, ILTR_ERR_INTERNAL_ERROR);

        if (ILTR_direction == ILTR_IMPORT)
        {
            //--- When importing use list of TARGET fields.
            pList = pMap->pTarget;
            count = pMap->nTarget + ILTR_pTableInfo->nExtraFields;
        }
        else
        {
            //--- When exporting use list of SOURCE fields.
            pList = pMap->pSource;

```

```

        count = pMap->nSource + ILTR_pTableInfo->nExtraFields;
    }
}
else
{
    /*-----
    * The standard un-inverted rule is that we look up the field in the
    * target field list if we're currently in Phase10 (exporting from
    * target) or in Phase30 (importing into target).
    *-----*/
    BOOLEAN bUseTargetList = (ILTR_phase == ILTR_PHASE10
                               || ILTR_phase == ILTR_PHASE30);

    /*-----
    * For inverted lookup simply reverse the standard rule.
    *-----*/
    if (bInvertedLookup)
        bUseTargetList = !bUseTargetList;

    if (bUseTargetList)
    {
        pList = pMap->pTarget;
        count = pMap->nTarget + ILTR_pTableInfo->nExtraFields;
    }
    else
    {
        pList = pMap->pSource;
        count = pMap->nSource + ILTR_pTableInfo->nExtraFields;
    }
}

//----- Scan the field list until we find the given field label.
for (i=0; i < count; i++)
{
    if (IL_STRINGS_CI_EQUAL(pList[i].IntName, label))
    {
        *attrs = pList[i].Attrs;
        *type = pList[i].Type;
        *pFldSize = pList[i].Width;
        if (pszTypeDesc != NULL)
        {
            IL_SAFE_STRINGCOPYN ( pszTypeDesc,
                                  pList[i].TypeDesc,
                                  ILTR_MAX_TYPEDESC );
        }
        return SUCCESS;
    }
}

//----- Reached end of list without finding required field label.
return FAILURE;
}

```

```

/*-----
* Name:      sst.c
* Part of:   IntelliLink translation harness (ILTR library)
* Contents:  SST (Section SubType) Tagging & Filtering Functions:
*
*           ILSST_AddTag    -- for TIF to call
*           ILSST_AddTag2  -- for ILTR\fldget.c to call
*           ILSST_Filter
*           ILSST_SubTypeOK
*           ILSST_StripTag
*
* Author:    David Boothby, Copyright (c) IntelliLink, 1996
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*        a "base" DLL rather than being statically linked into every
*        translator. Please ensure that all non-static functions in this
*        module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#include "iltr.h"
#include "ilx.h"
#include <ctype.h>

/*-----
* Name:    ILSST_AddTag
* Called from: ILTIF2\ILTIF.cpp\ILTIFGetField and from ILSST_AddTag2
*
* RETURNS:  SUCCESS or ILTR_ERR_TRUNC or error code for fatal abnormal error
*-----*/
ILBASEFN_int ILSST_AddTag
( ILTR_PTRANSL tr,
  INT32 SizeLimit,
  INT32 IL_DIST *pLength,  // length including null terminator byte
  IL_PSTR IL_DIST *ppText,
  IL_PSTR szTagDigits,
  IL_PSTR szTypeDesc )
{
    int rc, rc2;
    INT32 newFullSize, newActualSize;
    int copyBytes;
    char szTag[20];
    IL_PSTR p;

    //----- Construct full tag to use as suffix to "real" field value
    if (*pLength < 2)
        IL_SPRINTF(szTag, "{%.10s}", szTagDigits);
    else
        IL_SPRINTF(szTag, " {%.10s}", szTagDigits);

    //----- see whether adding tag will push us over the size limit
    newFullSize = *pLength + IL_STRLEN(szTag);
    if (newFullSize <= SizeLimit)
    {
        //----- plenty of room for tagged value
        newActualSize = newFullSize;
        copyBytes = (int) *pLength - 1;
        rc = SUCCESS;
    }
    else
    {
        //----- we'll have to do truncation!!
        newActualSize = SizeLimit;
        copyBytes = (int) (*pLength - 1L - (newFullSize - SizeLimit));
        rc = ILTR_ERR_TRUNC;
    }

    if (ILTR_pSSTBuf == NULL)
    {
        //----- initialize reusable buffer ILTR_pSSTBuf
        IL_ALLOC_MEM (sizeof (ILUT_BUFFER), ILTR_hSSTBuf, ILTR_pSSTBuf);
        if (ILTR_pSSTBuf == NULL)
            return ILERROR(sizeof(ILUT_BUFFER), ILTR_ERR_NOMEM);

        rc2 = ILUT_InitBuffer (ILTR_pSSTBuf, 0, 512, ILTR_MAX_FIELDLNGTH+1);
    }
}

```



```

        if (rc2 != SUCCESS)
            return ILERROR(rc2, ILTR_ERR_INTERNAL_ERROR);
    }

    //----- Ensure that 'SST' buffer is big enough. If not, expand it.
    rc2 = ILUT_GetBuffer (ILTR_pSSTBuf, newActualSize);
    if (rc2 != SUCCESS)
        return ILERROR(rc2, ILTR_ERR_NOMEM);

    p = (IL_PSTR) ILTR_pSSTBuf->pBuffer;
    IL_STRNCPY (p, *ppText, copyBytes);
    IL_STRCPY (p+copyBytes, szTag);

    *pLength = newActualSize;
    *ppText = p;

    return rc;
} //---- ILSST_AddTag

/*-----
* Name: ILSST_AddTag2
* Called from: ILTR\fldget.c
*
* RETURNS: SUCCESS or ILTR_ERR_TRUNC or error code for fatal abnormal error
*-----*/
ILBASEFN_int ILSST_AddTag2 // ADD TAG function for ILFldGet to call
(
    ILTR_PTRANSI tr,
    unsigned int bufSize,
    unsigned int IL_DIST *pLength, // length excluding NULL
    IL_PSTR lpBuffer,
    IL_PSTR szTagDigits,
    IL_PSTR szTypeDesc )
{
    int rc;
    INT32 len = (INT32) *pLength+1; // length including NULL
    IL_PSTR pData = lpBuffer;

    rc = ILSST_AddTag (tr, (INT32) bufSize, &len, &pData, szTagDigits, szTypeDesc);
    if (rc == SUCCESS || rc == ILTR_ERR_TRUNC)
    {
        *pLength = (unsigned int) len-1; // length excluding NULL
        IL_STRCPY (lpBuffer, pData);
    }

    return rc;
} //---- ILSST_AddTag2

/*-----
* ILSST_Filter -- decide whether or not to EXCLUDE current record
* Called from ILTR\EXPORT.C
* Returns:
*   SUCCESS if record is NOT to be excluded.
*   ILTR_SKIP_WRITE if record IS to be excluded.
*   other abnormal error codes if necessary
*-----*/
ILBASEFN_int ILSST_Filter (ILTR_PTRANSI tr)
{
    int rc;
    char szBuf[10];
    long len;

    //----- Don't exclude record if SST filtering is disabled
    if (ILTR_Flags & ILTR_DISABLE_SST_FILTERING)
        return SUCCESS;

    //----- apply special exclusions when synchronizing
    if (ILTR_nSynchronize)
    {
        //----- don't exclude record when synchronizing a 'TotalRebuild' system
        if (ILTR_nAttribs & ILTB_ATT_TOTAL_REBUILD)
            return SUCCESS;
    }

```

```

/*-----
 * For Fast Sync, don't check subtype for DELETE items.
 * Typically DELETE items are devoid of any data content,
 * including sub type information.
 *-----*/

//----- Retrieve value of _Delta field.
len = (long) sizeof(szBuf);
rc = ILTRGetField (tr, ILTR_FLD_DELTA, szBuf, &len);

//----- Don't check subtype if this is a DELETE
if (rc == SUCCESS && IL_STRINGS_EQUAL(szBuf, ILTR_DELTA_DELETE))
    return SUCCESS;
}

//----- Retrieve value of SUBTYPE field.
len = (long) sizeof(szBuf);
rc = ILTRGetField (tr, ILTR_SUB_TYPE, szBuf, &len);
if (rc != SUCCESS)
    return ILERROR (rc, ILTR_ERR_INTERNAL_ERROR);

//----- apply subtype matching rules
rc = ILSST_SubTypeOK (tr, szBuf);
switch (rc)
{
    case FALSE: ILTR_action = ILTR_ACT_FILTER;
                return ILTR_SKIP_WRITE;

    case TRUE:  return SUCCESS;
    default:   return rc;
}
} //---- ILSST_Filter

/*-----
 * ILSST_SubTypeOK -- apply subtype matching rules to decide whether record
 *                   subtype is suitable for transfer
 *                   to the current Target Section.
 * Called from ILIF2\TIFPUT.CPP\ComputeSearchKeyValues and from ILSST_Filter.
 * Returns:
 *   TRUE/FALSE if record subtype is OK/not OK
 *   other abnormal error codes if necessary
 *-----*/
ILBASEFN_int ILSST_SubTypeOK (ILTR_PTRANSL tr, IL_PSTR szSubType)
{
    int subtype;

    subtype = IL_atoi (szSubType);

    if (ILTR_TargetSST == ILX_SUBSECT_MAIN)
    {
        if (subtype == ILX_SUBSECT_MAIN)
            //----- good match: both record and target section have subtype==0
            return TRUE;
        else
        {
            int i;
            for (i=0; i < ILTR_TargetSSTCount; i++)
                if (subtype == ILTR_TargetSSTList[i])
                {
                    /*-----
                     * EXCLUDE this record on THIS translation pass. There is an
                     * an exact-match "home" for this record, in the Target App, so
                     * we don't want to put it into the "catch-all" MAIN section.
                     *-----*/
                    return FALSE;
                }

            // no exact-match home, so let record go into catch-all MAIN section
            return TRUE;
        }
    }

    else if (subtype == ILTR_TargetSST)
        //----- subtype of record matches subtype of target section
        return TRUE;
}

```

```

    else
        //----- subtype of record does not match subtype of target section
        return FALSE;
} //----- ILSST_SubTypeOK

/*-----
* Name: ILSST_StripTag
* Called from: ILTR\fldput.c and ILTIF2\ILTIF.cpp\ILTIFPutField
*
* NOTE: some day this function may get a lot fancier. Right
* now it only knows how to detect NUMERIC SUFFIX TAGS.
* The overall field formats that it knows about are
* just these two (shown by example):
*
*      "{9}"      -- no "real data", just a tag
*      "Fred {9}" -- real data "Fred" with tag suffix
*
* the "szTypeDesc" arg will someday prescribe other formats
*
* NOTE: trailing spaces after a tag are tolerated, and are STRIPPED.
*
* If a TAG is found it is stripped from the field that carries it,
* and is copied back into the caller's 'szTag' buffer. The caller
* is responsible for putting the tag value into the "_subType" field.
*
* if no TAG is found, szTag buffer is set to NULLSTRING.
*
* RETURNS: SUCCESS or error code for fatal abnormal error
*-----*/
ILBASEFN_int ILSST_StripTag
(
    ILTR_PTRANSI tr,
    IL_PSTR IL_DIST *ppText,
    unsigned int IL_DIST *pLen,
    IL_PSTR szTypeDesc,
    IL_PSTR szTag )    // char szTag[ILTR_MAX_TAG_LEN]
{
    IL_PSTR p;
    IL_PSTR pCloseCurly;
    IL_PSTR pOpenCurly;
    int tagDigits;

    IL_MAKE_STRING_NULL(szTag); // null tag, unless changed below...

    /*-----
    * Commented out this next check - it was messing us up in ILX_V3
    * SmartMerge Import from a nonMain section into a MAIN section.
    * The PutProcessing code, of the Target xlator, uses ILFldGet to
    * read from ILIF file. Tagged field gets decorated. Then xlator
    * calls ILTIFPutField. We want decorations stripped back off again.
    * NOTE: with this check disabled we can "see" tags when exporting
    * from nonMain sections. This was not really intended, but isn't
    * harmful. Such tags should only get there due to user deviousness.
    * IntelliLink will never put tags into nonMain sections.
    *-----*/
    //----- If we're NOT working with data from a MAIN section, do nothing
    // if (ILTR_SourceSST != ILX_SUBSECT_MAIN)
    //     return SUCCESS;

    //----- If TAGGING is disabled, do nothing
    if (ILTR_Flags & ILTR_DISABLE_SST_TAGGING)
        return SUCCESS;

    /*-----
    * search for suffix tag in curly braces; if we don't find one, this
    * function is a happy little NO-OP.
    *-----*/

    //----- find LAST closing curly brace
    pCloseCurly = IL_STRCHR (*ppText, '}');
    if (pCloseCurly == NULL)
        //----- no curly brace means no tag, hence NO OP
        return SUCCESS;

    //----- check for non-SPACE char after closing brace

```

```

p = pCloseCurly+1;
while (*p != 0)
    if (*p++ != ' ')
        //---- found something; must be no tag.  hence NO OP
        return SUCCESS;

//---- find LAST opening curly brace
pOpenCurly = IL_STRCHR (*ppText, '{');
if (pOpenCurly == NULL)
    //---- no curly brace means no tag, hence NO OP
    return SUCCESS;

//---- demand that there be a SPACE between the real data and the tag
if ((pOpenCurly > *ppText) && (*(pOpenCurly-1) != ' '))
    //---- no SPACE means no tag, hence NO OP
    return SUCCESS;

tagDigits = (pCloseCurly-pOpenCurly) - 1;

if (tagDigits > ILTR_MAX_TAG_LEN || tagDigits < 1)
    //---- too much or too little "meat" between the braces; not a valid tag
    return SUCCESS;

//---- make sure the tag is 100% numeric
p = pOpenCurly+1;
while (p < pCloseCurly)
    if (!isdigit(*p++))
        //---- tag isn't all numeric; isn't a valid tag; NO-OP
        return SUCCESS;

/*-----
 * Tag Validation is complete.  Copy the "meat" of the tag into the
 * caller's buffer
 *-----*/
IL_STRNCPY (szTag, pOpenCurly+1, tagDigits);
szTag[tagDigits] = 0;

/*-----
 * NOTE: we never need to copy the data into another buffer.
 * For SUFFIX tags, we simply reduce the LENGTH of the field value;
 * for PREFIX tags, shift the text pointer AND reduce the length.
 *-----*/
if (pOpenCurly == *ppText)
    *pLen = 0;
else
    *pLen = pOpenCurly - *ppText - 1;

return SUCCESS;
} //---- ILSST_Striptag

```

```

/*-----
* Name:      ILRPT.H
* Purpose: Header file used with repeating items
* Author:   Mike Blanchette, Copyright (c) IntelliLink Corporation, 1993
*-----*/
#ifndef __ILRPT
#define __ILRPT // Only include header once
                // Signal header inclusion

//----- Common Intellilink types
#include "iltypes.h"

//----- Miscellaneous constants
#define ILTR_MAX_FLDNAME 31 // Max size of field names
#define ILTR_REP_BASIC   "_repBasic" // Basic repeat field
#define ILTR_REP_XDATE   "_repExcl" // Exclusion date field
#define ILTR_APP_DATA    "_appData" // Application binary field

//----- Days of the week (use INT16, not enum, to be size-invariant)
typedef INT16 ILTR_DAY;
#define ILTR_SUN 1 // Sunday
#define ILTR_MON 2 // Monday
#define ILTR_TUE 3 // Tuesday
#define ILTR_WED 4 // Wednesday
#define ILTR_THU 5 // Thursday
#define ILTR_FRI 6 // Friday
#define ILTR_SAT 7 // Saturday
//----- Next 3 values used with xxx_AND_CLASS repeat patterns only
#define ILTR_ANY_DAY 8 // any day of the week
#define ILTR_ANY_WEEKDAY 9 // any 1 of Mon-Fri
#define ILTR_ANY_WEEKEND_DAY 10 // Saturday or Sunday

//----- Months of the year (use INT16, not enum, to be size-invariant)
typedef INT16 ILTR_MONTH;
#define ILTR_MONTH_JAN 1
#define ILTR_MONTH_FEB 2
#define ILTR_MONTH_MAR 3
#define ILTR_MONTH_APR 4
#define ILTR_MONTH_MAY 5
#define ILTR_MONTH_JUN 6
#define ILTR_MONTH_JUL 7
#define ILTR_MONTH_AUG 8
#define ILTR_MONTH_SEP 9
#define ILTR_MONTH_OCT 10
#define ILTR_MONTH_NOV 11
#define ILTR_MONTH_DEC 12

//----- Supported repetition types (use INT16, not enum, to be size-invariant)
typedef INT16 ILTR_REPEATTYPE;
#define ILTR_NOREPEAT 0 // No repeat
#define ILTR_DAILY 1 // Daily
#define ILTR_WEEKLY 2 // Weekly
#define ILTR_MONTHLY 3 // Monthly
#define ILTR_MONTHLY_BY_POS 4 // Monthly by position
#define ILTR_YEARLY 5 // Yearly
#define ILTR_YEARLY_BY_POS 6 // Yearly by position
#define ILTR_MIXED 7 // Mixed
#define ILTR_MIXED_BY_POS 8 // Mixed by position
#define ILTR_EXCLUSION 9 // Exclusion
#define ILTR_MONTHLY_BY_POS_AND_CLASS 10 // e.g. 1st weekday of each month
#define ILTR_YEARLY_BY_POS_AND_CLASS 11 // e.g. 1st weekday of May
#define ILTR_WEEKLY_DAYS 12 // e.g. every Mon,Wed,Fri
#define ILTR_QUARTERLY 13 // Quarterly
#define ILTR_QUARTERLY_BY_POS 14 // Quarterly by position

//----- if you add repeat types, please update the TEST program!

/*-----
* NOTE: when using the Repeat Types ILTR_MONTHLY_BY_POS_AND_CLASS and
*       ILTR_YEARLY_BY_POS_AND_CLASS, you must set "dayOfWeek" to one
*       of ILTR_ANY_DAY, ILTR_ANY_WEEKDAY, or ILTR_ANY_WEEKEND_DAY,
*       and you must set "day" to any one of the ILTR_POSITION_IN_MONTH
*       enum values.
*-----*/

//----- Weeks of the month (use INT16, not enum, to be size-invariant)

```

```

typedef INT16 ILTR_WEEK;
#define ILTR_WEEK_1ST          1          // First week
#define ILTR_WEEK_2ND          2          // Second week
#define ILTR_WEEK_3RD          3          // Third week
#define ILTR_WEEK_4TH          4          // Fourth week
#define ILTR_WEEK_LAST         5          // Last week

//----- Weeks of the quarter (use INT16, not enum, to be size-invariant)
typedef INT16 ILTR_WEEK;
#define ILTR_QUARTER_WEEK_1ST  1          // First week
#define ILTR_QUARTER_WEEK_2ND  2          // Second week
#define ILTR_QUARTER_WEEK_3RD  3          // Third week
#define ILTR_QUARTER_WEEK_4TH  4          // Fourth week
#define ILTR_QUARTER_WEEK_5TH  5          // Fifth week
#define ILTR_QUARTER_WEEK_6TH  6          // Sixth week
#define ILTR_QUARTER_WEEK_7TH  7          // Seventh week
#define ILTR_QUARTER_WEEK_8TH  8          // Eighth week
#define ILTR_QUARTER_WEEK_9TH  9          // Ninth week
#define ILTR_QUARTER_WEEK_10TH 10         // Tenth week
#define ILTR_QUARTER_WEEK_11TH 11         // Eleventh week
#define ILTR_QUARTER_WEEK_12TH 12         // Twelfth week
#define ILTR_QUARTER_WEEK_13TH 13         // Thirteenth week
#define ILTR_QUARTER_WEEK_LAST 14         // Last week

//----- Artificial "last" values
#define ILTR_MONTH_DAY_LAST    32         // Last day of month
#define ILTR_QUARTER_DAY_LAST  99         // Last day of quarter

/*-----
 * NOTE: the following enum values are only used with the
 * Repeat Types ILTR_MONTHLY_BY_POS_AND_CLASS and
 * ILTR_YEARLY_BY_POS_AND_CLASS. Put in the "day" field.
 *-----*/
typedef INT16 ILTR_POSITION_IN_MONTH;
#define ILTR_POS_1ST          1
#define ILTR_POS_2ND          2
#define ILTR_POS_3RD          3
#define ILTR_POS_4TH          4
#define ILTR_POS_LAST         5

//----- Pointer to numeric date field
typedef long IL_DIST *ILTR_PDATES;

//----- Repeating item structure
typedef struct _repeat
{
    /*-----
     * Default values must be supplied for all common fields below.
     * Observe the following rules in using the structure:
     * (1) All dates are expressed as the number of days since 1/1/1900
     * (2) Set stopDate to -1 if no stop date exists (indefinite item)
     * (3) Set numExDates to 0 and exDates to NULL if no exclusions exist
     * (4) Memory for exDates must be allocated by the caller
     *-----*/

    //----- Common fields for ALL repeat types
    ILTR_REPEATTYPE type;          // Repeat type
    long startDate;                // Duration start date
    long stopDate;                 // Duration stop date
    INT16 frequency;               // Repeat frequency (every n)
    INT16 numDays;                 // Consecutive days to repeat
    char startField[ILTR_MAX_FLDNAME]; // Name of start date field
    char stopField[ILTR_MAX_FLDNAME]; // Name of stop date field

    /*-----
     * Variable portion of structure. Values must be provided in fields
     * listed below based on the indicated repeat type:
     * ILTR_NOREPEAT: (None but numDays field may be set)
     * ILTR_DAILY: (None)
     * ILTR_WEEKLY: dayOfWeek
     * (NOTE: for an item that occurs every weekday,
     * use Type=WEEKLY, dayOfWeek=MONDAY, numDays=5)
     * ILTR_MONTHLY: day
     * ILTR_MONTHLY_BY_POS: dayOfWeek, weekOfMonth
     * ILTR_YEARLY: month, day
     *-----*/

```

```

*   ILTR_YEARLY_BY_POS:      dayOfWeek, month, weekOfMonth
*   ILTR_MIXED:             day, months
*   ILTR_MIXED_BY_POS:      days, months, weeks
*   ILTR_MONTHLY_BY_POS_AND_CLASS: day, dayOfWeek
*   ILTR_YEARLY_BY_POS_AND_CLASS: day, dayOfWeek, month
*   ILTR_WEEKLY_DAYS:       days
*   ILTR_QUARTERLY:         day
*   ILTR_QUARTERLY_BY_POS   dayOfWeek, weekOfMonth (alias weekOfQuarter)
*-----*/
INT16 day;                      // Day of month (1-31)
INT16 month;                    // Month of year (1-12)
ILTR_DAY dayOfWeek;            // Day of week (Sun-Sat)
ILTR_WEEK weekOfMonth;         // Week of month (1-5)
BOOL16 days[7];                // Day of week (TRUE or FALSE)
BOOL16 months[12];             // Months (TRUE or FALSE)
BOOL16 weeks[5];               // Week of month (TRUE or FALSE)

/*-----*/
* This information must be the last in this structure.  Because Repeat
* information is passed between the translators, and different trans-
* lators use different memory models, exDates may be either a near or
* a far value.  The value dummy is provided to account for cases where
* the source translator uses near and the target uses far (a near value
* goes into the intermediate file, and the dummy goes into the
*-----*/
INT16 numExDates;               // Count of exclusion dates
IL_HANDLE hExDates;            // Handle to exclusion list
IL_HANDLE_PADDING(xx)          // pad if IL_HANDLE is less than 32 bits
ILTR_PDATES exDates;           // Pointer to exclusion list
INT16 dummy;                   // Account for near/far mismatch
} ILTR_REPEAT;

/*-----*/
* Max Fanout counts:  the following structure is used to hold a set of
* maxima which apply when fanning repeating items.
* Different limits apply to different repeat types.  A negative
* count is used to make the maximum vary according to the numDays count
* (or for ILTR_WEEKLY_DAYS to vary according to the number of days
* selected per week).  For example if weekly limit is -52 then
* a WEEKLY_DAYS pattern for Mondays, Wednesdays, and Fridays will be
* fanned out for 52 weeks, for a total of 156 instances.  A positive
* count gives a simple rigid limit, with no "numDays" dependency.
*
* The commonly used maxima structure is ILTR_FanoutMaxima.  ILTR.H
* has access macros for each of the members of ILTR_FanoutMaxima.
*
* At the start of each translation phase (hence independently for source
* and target translators) the fanout limits are set to "reasonable"
* default values.  Some translators may override the defaults by
* putting different values into the ILTR_FanoutMaxima structure when
* the DataStore's "Open" function is called.
*-----*/
typedef struct
{
    INT16 nDailyMaxFanout;       // max for DAILY patterns
    INT16 nWeeklyMaxFanout;      // max for WEEKLY & WEEKLY_DAYS
    INT16 nMonthlyMaxFanout;     // max for all 3 MONTHLY patterns
    INT16 nQuarterlyMaxFanout;   // max for QUARTERLY patterns
    INT16 nYearlyMaxFanout;      // max for all 3 YEARLY patterns
    INT16 nOtherMaxFanout;       // max for all other patterns
}
ILTR_FANOUT_MAXIMA, IL_DIST *ILTR_PFANOUT_MAXIMA;

#endif // __ILRPT

```

```

/*-----
* Name:      ILRepeatNextDate
* Purpose:   Get next valid date for repeating item
* Input:     Pointer to repeat structure and start date field
* Return:    SUCCESS or error code
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*         a "base" DLL rather than being statically linked into every
*         translator. Please ensure that all non-static functions in this
*         module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#include "iltr.h"
#include <time.h>

//----- Function prototypes.
static int isLast ( long );
static int normalizeDay ( int,
                        int,
                        int );
static void normalizeDate ( int IL_DIST *,
                        int IL_DIST *,
                        int IL_DIST *,
                        int IL_DIST * );

static void goBackward ( ILTR_PREPEAT repeat,
                        long IL_DIST *nDate, int nDay );

static void goForward ( ILTR_PREPEAT repeat,
                        long IL_DIST *nDate, int nDay );

static int nextDaily (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextWeekly (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextWeeklyDays (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextMonthly (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextYearly (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextMixed (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextMixedByPos (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextByPos (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextByPosAndClass (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextQuarterly (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);
static int nextQuarterlyByPos (ILTR_PREPEAT, long IL_DIST *, long IL_DIST *);

//-----
ILBASEFN_int ILRepeatNextDate ( ILTR_PREPEAT repeat,
                                long IL_DIST *nDate,
                                long IL_DIST *priorDate )
{
    //----- please use & maintain the test program in iltr\test !!!!

    if (repeat->type != ILTR_NO_REPEAT)
    {
        if ( (repeat->startDate < 0)
            || (repeat->stopDate < -1)
            || (repeat->frequency < 1)
            || (repeat->numDays < 1) )
            return ILERROR (ILTR_ERR_REPEAT, ILTR_ERR_REPEAT);
    }

    switch (repeat->type)
    {
        case ILTR_DAILY:      return nextDaily (repeat, nDate, priorDate);
        case ILTR_WEEKLY:    return nextWeekly (repeat, nDate, priorDate);
        case ILTR_WEEKLY_DAYS:
                                return nextWeeklyDays (repeat, nDate, priorDate);
        case ILTR_MONTHLY:   return nextMonthly (repeat, nDate, priorDate);
        case ILTR_YEARLY:    return nextYearly (repeat, nDate, priorDate);
        case ILTR_MIXED:     return nextMixed (repeat, nDate, priorDate);
        case ILTR_MONTHLY_BY_POS:
        case ILTR_YEARLY_BY_POS:
                                return nextByPos (repeat, nDate, priorDate);
    }
}

```



```

    case ILTR_MIXED_BY_POS:
        return nextMixedByPos (repeat, nDate, priorDate);
    case ILTR_MONTHLY_BY_POS_AND_CLASS:
    case ILTR_YEARLY_BY_POS_AND_CLASS:
        return nextByPosAndClass (repeat, nDate, priorDate);
    case ILTR_QUARTERLY:
    case ILTR_QUARTERLY_BY_POS:
        return nextQuarterlyByPos (repeat, nDate, priorDate);
    case ILTR_NOREPEAT:
        *priorDate = *nDate;    // no change from previous version...
        return SUCCESS;        // should this be tightened up?

    case ILTR_EXCLUSION:        // inappropriate repeat type
    default:                    // unknown repeat type !!
        return ILERROR (ILTR_ERR_REPEAT, ILTR_ERR_REPEAT);
}
}

//-----//
static int nextDaily ( ILTR_PREPEAT repeat,
                      long IL_DIST *nDate,
                      long IL_DIST *priorDate )
{
    //----- First valid date IS the start date.
    if (*priorDate == -1)
        *nDate = repeat->startDate;

    //----- Increment date by frequency number.
    else *nDate = *priorDate + repeat->frequency;

    *priorDate = *nDate;
    return SUCCESS;
}

//-----//
static int nextWeekly ( ILTR_PREPEAT repeat,
                      long IL_DIST *nDate,
                      long IL_DIST *priorDate )
{
    //----- Compute first date having specified day of week (Sun-Sat).
    if (*priorDate == -1)
    {
        int nDay = IL_DayOfWeek (repeat->startDate);
        *nDate = repeat->startDate + repeat->dayOfWeek - nDay;
        if (nDay > repeat->dayOfWeek)
            *nDate += 7;
    }

    //----- Increment date by number of weeks specified by frequency.
    else *nDate = *priorDate + (repeat->frequency * 7);

    *priorDate = *nDate;
    return SUCCESS;
}

/*-----
* Name:      chooseNextDay
* Called by: nextWeeklyDays (only caller)
* Purpose:   scan through the "days" array for next day.
* Examples:  if days[] says MonWedFri, and nPreviousDay is Mon or Tue
*            choose Wednesday; if nPreviousDay is Fri Sat or Sun,
*            choose Monday.
*-----*/

//-----//
static int chooseNextDay (int nPreviousDay, BOOL16 *days)
{
    int i, dayIndex;

    dayIndex = nPreviousDay-1; // get zero-based index from 1-based enum
    for (i=0; i < 7; i++)

```

```

    {
        dayIndex += 1;                // advance to next day
        if (dayIndex > 6)              // after Saturday...
            dayIndex = 0;              // ...is Sunday (next week)
        if (days[dayIndex])
            return (dayIndex+1);
    }
    return (-1);    // ERROR ... no true days[]
}

//-----//
static int nextWeeklyDays (ILTR_PREPEAT repeat,
                          long IL_DIST *plDate,
                          long IL_DIST *plPriorDate )
{
    long lPreviousDate;
    int nPreviousDay;
    int nThisDay;

    lPreviousDate = *plPriorDate;
    if (lPreviousDate == -1)
        lPreviousDate = repeat->startDate - 1;
    nPreviousDay = IL_DayOfWeek (lPreviousDate);

    nThisDay = chooseNextDay (nPreviousDay, repeat->days);
    if (nThisDay == -1)
        return ILERROR (ILTR_ERR_REPEAT, ILTR_ERR_REPEAT);

    //----- Now figure DATE for chosen DAY.

    *plDate = lPreviousDate + (nThisDay - nPreviousDay);
    if (nThisDay < nPreviousDay)
    {
        // e.g. This is Monday; Previous is Friday
        if (*plPriorDate == -1)
            *plDate += 7;          // for FIRST occurrence, use date in the week
        //----- right after the startDate (ignore FREQUENCY)
    }
    else
        *plDate += (repeat->frequency * 7); // go forward N weeks

    *plPriorDate = *plDate;
    return SUCCESS;
}

//-----//
static int nextMonthly ( ILTR_PREPEAT repeat,
                        long IL_DIST *nDate,
                        long IL_DIST *priorDate )
{
    int nDay      = 0;                // Day of week (Sun-Sat)
    int nWeek     = 0;                // Week of month (1-5)
    int month, day, year;              // Date fields
    long holdDate;                    // Encoded date

    /*-----
    * Compute next date having the specified day of month (1-31).
    * Decode start date on initial call. On subsequent calls,
    * use the prior date field.
    *-----*/
    if (*priorDate == -1)
        holdDate = repeat->startDate;
    else holdDate = *priorDate;
    IL_DateDecode (holdDate, &month, &day, &year);

    /*-----
    * Iterate until we have a date that exceeds the start date.
    * By definition, this loop can never iterate more than twice.
    *-----*/
    if (*priorDate == -1)
    do
    {
        day = normalizeDay (repeat->day, month, year);
        IL_DateEncode (month, day, year, nDate);
    }
}

```

```

        month++;
        normalizeDate (&year, &month, &nWeek, &nDay);
    }
    while (*nDate < repeat->startDate);

    //----- Increment month based on setting of frequency field.
    else
    {
        month += repeat->frequency;
        normalizeDate (&year, &month, &nWeek, &nDay);
        day = normalizeDay (repeat->day, month, year);
        IL_DateEncode (month, day, year, nDate);
    }

    *priorDate = *nDate;
    return SUCCESS;
}

//-----//
static int nextByPos ( ILTR_PREPEAT repeat,
                      long IL_DIST *nDate,
                      long IL_DIST *priorDate )
{
    int nDay      = 0;           // Day of week (Sun-Sat)
    int nDays     = 0;           // Days in month (1-31)
    int nWeek     = 0;           // Week of month (1-5)
    int holdDay   = 0;           // Temporary day of week
    int holdWeek  = 0;           // Week of month (1-5)
    int month, day, year;        // Date fields
    long holdDate;               // Encoded date

    /*-----
    * Begin with start date on initial call. Use the prior date
    * field on subsequent calls.
    *-----*/
    if (*priorDate == -1)
        holdDate = repeat->startDate;
    else holdDate = *priorDate;
    IL_DateDecode (holdDate, &month, &day, &year);

    //----- Increment month or year on subsequent calls.
    if (*priorDate != -1)
        if (repeat->type == ILTR_MONTHLY_BY_POS)
        {
            month += repeat->frequency;
            normalizeDate (&year, &month, &nWeek, &nDay);
        }
        else year += repeat->frequency;

    /*-----
    * Compute next date in specified week of month and on given
    * day of week. Start from the end of month and work backwards
    * to locate the desired date.
    *-----*/
    if (repeat->type == ILTR_YEARLY_BY_POS)
        month = repeat->month;

    /*-----
    * This loop computes the date of given repeat item for
    * the starting month and again for the succeeding month
    * IFF the first date precedes the prior date.
    * By definition, the loop can never iterate more than twice.
    *-----*/
    do
    {
        /*-----
        * Find the last day of the month and locate the last
        * occurrence of the desired day of week.
        *-----*/
        nDays = IL_DaysInMonth (month, year);
        IL_DateEncode (month, nDays, year, nDate);
        nDay = IL_DayOfWeek (*nDate);
        if (nDay < repeat->dayOfWeek)
            *nDate = *nDate - 7 + (repeat->dayOfWeek - nDay);
    }
}

```

```

    else
        *nDate = *nDate - (nDay - repeat->dayOfWeek);

    /*-----
    * Now adjust the computed date to correspond to the
    * desired week of month.
    *-----*/
    nWeek = IL_WeekInMonth (*nDate);
    if (repeat->weekOfMonth < nWeek)
        *nDate -= (7 * (nWeek - repeat->weekOfMonth));

    //----- Increment the month or year in case we iterate again.
    if (repeat->type == ILTR_YEARLY_BY_POS)
        year++;
    else month++;
    normalizeDate (&year, &month, &nWeek, &nDay);
}
while (*nDate < repeat->startDate);

*priorDate = *nDate;
return SUCCESS;
}

//-----//
static int nextYearly ( ILTR_PREPEAT repeat,
                        long IL_DIST *nDate,
                        long IL_DIST *priorDate )
{
    int month, day, year;           // Date fields

    /*-----
    * Decode start date on initial call.  On subsequent calls,
    * use the prior date field.
    *-----*/
    if (*priorDate == -1)
    {
        IL_DateDecode (repeat->startDate, &month, &day, &year);

        /*-----
        * This loop can never iterate more than twice.  It computes
        * the yearly date and only iterates a second time if the
        * first computed date precedes the item start date.
        *-----*/
        do
        {
            day = normalizeDay (repeat->day, repeat->month, year);
            IL_DateEncode (repeat->month, day, year++, nDate);
        } while (*nDate < repeat->startDate);
    }
    else
    {
        //----- Increment year based on setting of frequency field.
        IL_DateDecode (*priorDate, &month, &day, &year);
        year += repeat->frequency;
        day = normalizeDay (repeat->day, month, year);
        IL_DateEncode (month, day, year, nDate);
    }

    *priorDate = *nDate;
    return SUCCESS;
}

//-----//
static int nextMixed ( ILTR_PREPEAT repeat,
                       long IL_DIST *nDate,
                       long IL_DIST *priorDate )
{
    int nDay      = 0;           // Day of week (Sun-Sat)
    int nWeek     = 0;           // Week of month (1-5)
    int holdDay   = 0;           // Temporary day of week
    int month, day, year;        // Date fields
    long holdDate;               // Encoded date

```

```

/*-----
 * Compute next date having the specified month and day of month.
 * Decode start date on initial call. On subsequent calls,
 * use the prior date field.
 *-----*/
if (*priorDate == -1)
    holdDate = repeat->startDate;
else holdDate = *priorDate;
IL_DateDecode (holdDate, &month, &day, &year);

//----- Increment month based on setting of frequency field.
if (*priorDate != -1)
{
    month++;
    normalizeDate (&year, &month, &nWeek, &nDay);
    day = normalizeDay (repeat->day, month, year);
}

/*-----
 * Iterate until we have a date that exceeds the start date.
 * The loop stops when the date exceeds the start date AND
 * corresponds to one of the checked months.
 *-----*/
while (TRUE)
{
    //----- Is month checked in list? Skip it if not in list.
    normalizeDate (&year, &month, &nWeek, &nDay);
    if (repeat->months[month-1] == FALSE)
    {
        month++;
        continue;
    }

    //----- Is the date in range?
    day = normalizeDay (repeat->day, month, year);
    IL_DateEncode (month, day, year, nDate);
    if (*nDate < repeat->startDate)
    {
        month++;
        continue;
    }

    //----- Found valid date. Stop the loop.
    break;
}
*priorDate = *nDate;
return SUCCESS;
}

//-----//
static int nextMixedByPos ( ILTR_PREPEAT repeat,
                           long IL_DIST *nDate,
                           long IL_DIST *priorDate )
{
    int nDay      = 0;           // Day of week (Sun-Sat)
    int nWeek     = 0;           // Week of month (1-5)
    int holdDay   = 0;           // Temporary day of week
    int holdWeek  = 0;           // Week of month (1-5)
    int month, day, year;        // Date fields
    int okToProcess;             // Process flag

    /*-----
     * Find first date in specified list of months, weeks, and
     * days. Start by computing the week number and day of week
     * for the starting date.
     *-----*/
    if (*priorDate == -1)
        *nDate = repeat->startDate;
    else *nDate = *priorDate + 1;

    /*-----
     * This loop iterates until a date is found which meets
     * the month, week, and day specification.
     *-----*/

```

```

while (TRUE)
{
    /*-----
    * Is this month checked in the list?
    * If not, skip it and go on to the next month.
    *-----*/
    IL_DateDecode (*nDate, &month, &day, &year);
    if (repeat->months[month-1] == FALSE)
    {
        month++;
        normalizeDate (&year, &month, &nWeek, &nDay);
        IL_DateEncode (month, 1, year, nDate);
        continue;
    }

    /*-----
    * Is this week checked in the list?
    * If not, skip it and go on to the next week.
    *-----*/
    holdWeek = IL_WeekInMonth (*nDate);

    /*-----
    * Accept date if this is the last occurrence of
    * the day this month and LAST week has been specified.
    *-----*/
    if (repeat->weeks[holdWeek-1] == FALSE)
    {
        okToProcess = 0;
        if (holdWeek == ILTR_WEEK_4TH)
            if (repeat->weeks[ILTR_WEEK_LAST-1] == TRUE)
                if (isLast (*nDate))
                    okToProcess++;

        /*----- Accept the date?
        if (!okToProcess)
        {
            (*nDate)++;
            continue;
        }
    }

    /*-----
    * Is this day of week checked in the list?
    * If not, skip it and go on to the next day.
    *-----*/
    holdDay = IL_DayOfWeek (*nDate);
    if (repeat->days[holdDay-1] == FALSE)
    {
        (*nDate)++;
        continue;
    }

    /*----- Found a date. Stop the loop.
    break;
}

/*----- Set prior date field and return successfully.
*priorDate = *nDate;
return SUCCESS;
}

//-----//
static int nextByPosAndClass ( ILTR_PREPEAT repeat,
                              long IL_DIST *nDate,
                              long IL_DIST *priorDate )
{
    int nDay      = 0;           // Day of week (Sun-Sat)
    int nDays     = 0;           // Days in month (1-31)
    int nWeek     = 0;           // Week of month (1-5)
    int holdDay   = 0;           // Temporary day of week
    int holdWeek  = 0;           // Week of month (1-5)
    int month, day, year;        // Date fields
    long holdDate;               // Encoded date
    int position;

```

```

int dayClass;

position = repeat->day;
if ((position < ILTR_POS_1ST) || (position > ILTR_POS_LAST))
    return ILERROR (ILTR_ERR_REPEAT, ILTR_ERR_REPEAT);

dayClass = repeat->dayOfWeek;
if ( (dayClass != ILTR_ANY_DAY)
    &&(dayClass != ILTR_ANY_WEEKDAY)
    &&(dayClass != ILTR_ANY_WEEKEND_DAY) )
    return ILERROR (ILTR_ERR_REPEAT, ILTR_ERR_REPEAT);

/*-----
 * Begin with start date on initial call. Use the prior date
 * field on subsequent calls.
 *-----*/
if (*priorDate == -1)
    holdDate = repeat->startDate;
else holdDate = *priorDate;
IL_DateDecode (holdDate, &month, &day, &year);

//----- Increment month or year on subsequent calls.
if (*priorDate != -1)
    if (repeat->type == ILTR_MONTHLY_BY_POS_AND_CLASS)
    {
        month += repeat->frequency;
        normalizeDate (&year, &month, &nWeek, &nDay);
    }
    else year += repeat->frequency;

if (repeat->type == ILTR_YEARLY_BY_POS_AND_CLASS)
    month = repeat->month;

/*-----
 * This loop computes the date of given repeat item for
 * the starting month and again for the succeeding month
 * IFF the first date precedes the prior date.
 * By definition, the loop can never iterate more than twice.
 *-----*/
do
{
    if (position == ILTR_POS_LAST)
    {
        //----- get date and day of last day of month as starting point
        nDays = IL_DaysInMonth (month, year);
        IL_DateEncode (month, nDays, year, nDate);
        nDay = IL_DayOfWeek (*nDate);
        goBackward (repeat, nDate, nDay);
    }
    else
    {
        //----- get date and day of first day of month as starting point
        IL_DateEncode (month, 1, year, nDate);
        nDay = IL_DayOfWeek (*nDate);
        goForward (repeat, nDate, nDay);
    }

    //----- Increment the month or year in case we iterate again.
    if (repeat->type == ILTR_YEARLY_BY_POS_AND_CLASS)
        year++;
    else month++;
    normalizeDate (&year, &month, &nWeek, &nDay);
}
while (*nDate < repeat->startDate);

*priorDate = *nDate;
return SUCCESS;
}

//-----////////////////////////////////////
static int nextQuarterly ( ILTR_PREPEAT repeat,
                          long IL_DIST *nDate,
                          long IL_DIST *priorDate )
{

```

```

int nDay      = 0;                // Day of week (Sun-Sat)
int nWeek     = 0;                // Week of month (1-5)
int month, day, year;             // Date fields
int tmpMonth, tmpDay, tmpYear;    // Temp date fields
long holdDate;                    // Encoded date

/*-----*/
* Compute next date having the specified day of month (1-31).
* Decode start date on initial call. On subsequent calls,
* use the prior date field.
*-----*/
if (*priorDate == -1)
    holdDate = repeat->startDate;
else holdDate = *priorDate;

//----- Compute first date of this quarter and decode
IL_DateDecode (holdDate, &month, &day, &year);
month = 1 + (((month - 1) / 3) * 3);
day = 1;
IL_DateEncode (month, day, year, &holdDate);

//----- Now decode it
IL_DateDecode (holdDate, &month, &day, &year);

/*-----*/
* Iterate until we have a date that exceeds the start date.
*-----*/
if (*priorDate == -1)
do
{
    IL_DateEncode (month, day, year, nDate);
    *nDate += repeat->day - 1;    // Compute actual date in quarter

    //----- Fudge for "last" day of quarter
    IL_DateDecode (*nDate, &tmpMonth, &tmpDay, &tmpYear);
    while (tmpMonth < month ||    // Back up Jan into December
           tmpMonth > month+2)    // ...others
    {
        (*nDate)--;
        IL_DateDecode (*nDate, &tmpMonth, &tmpDay, &tmpYear);
    }

    month += 3;
    normalizeDate (&year, &month, &nWeek, &nDay);
}
while (*nDate < repeat->startDate);

//----- Increment month based on setting of frequency field.
else
{
    month += 3;
    normalizeDate (&year, &month, &nWeek, &nDay);
    IL_DateEncode (month, day, year, nDate);
    *nDate += repeat->day - 1;    // Compute actual date in quarter

    //----- Fudge for "last" day of quarter
    IL_DateDecode (*nDate, &tmpMonth, &tmpDay, &tmpYear);
    while (tmpMonth < month ||    // Back up Jan into December
           tmpMonth > month+2)    // ...others
    {
        (*nDate)--;
        IL_DateDecode (*nDate, &tmpMonth, &tmpDay, &tmpYear);
    }
}

*priorDate = *nDate;
return SUCCESS;
}

//-----//
static int nextQuarterlyByPos ( ILTR_PREPEAT repeat,
                                long IL_DIST *nDate,
                                long IL_DIST *priorDate )
{

```



```

int holdDay    = 0;                // Temporary day of week
int month, day, year;              // Date fields
long holdDate;                     // Encoded date

/*-----
 * Begin with start date on initial call.  Use the prior date
 * field on subsequent calls.
 *-----*/
if (*priorDate == -1)
    holdDate = repeat->startDate;
else holdDate = *priorDate;

//----- Compute first date of this quarter and encode
IL_DateDecode (holdDate, &month, &day, &year);
month = 1 + (((month - 1) / 3) * 3);
day = 1;

//----- Make sure that desired date is in range
if (*priorDate == -1)
{
    //----- Skip to desired week
    if (repeat-> weekOfMonth == ILTR_QUARTER_WEEK_LAST)
    {
        int nTempMonth, nTempYear;

        //----- Go to first week of next quarter
        nTempMonth = month + 3;
        nTempYear = year + 3;
        normalizeDate (&nTempYear, &nTempMonth, NULL, NULL);
        IL_DateEncode (nTempMonth, 1, nTempYear, &holdDate);

        //----- Back up to desired day
        holdDate--;
        while (IL_DayOfWeek (holdDate) != repeat-> dayOfWeek)
            holdDate--;
    }
    else
    {
        //----- Goto first day of the quarter
        IL_DateEncode (month, day, year, &holdDate);
        while (IL_DayOfWeek (holdDate) != repeat-> dayOfWeek)
            holdDate++;

        //----- Advance by specified number of weeks
        holdDate += 7 * (repeat-> weekOfMonth - 1);
    }

    //----- Only increment if outside range
    if (holdDate < repeat-> startDate)
        month += 3;
}

//----- Increment quarter
else
    month += 3;

//----- Fix values if needed
normalizeDate (&year, &month, NULL, NULL);

//----- Skip to desired week
if (repeat-> weekOfMonth == ILTR_QUARTER_WEEK_LAST)
{
    //----- Go to first week of next quarter
    month = month + 3;
    normalizeDate (&year, &month, NULL, NULL);
    IL_DateEncode (month, 1, year, &nDate);

    //----- Back up to desired day
    (*nDate)--;
    while (IL_DayOfWeek (*nDate) != repeat-> dayOfWeek)
        (*nDate)--;
}
else
{
    //----- Now find the first day of the quarter

```

```

    IL_DateEncode (month, day, year, nDate);
    while (IL_DayOfWeek (*nDate) != repeat-> dayOfWeek)
        (*nDate)++;

    //----- Advance by specified number of weeks
    *nDate += 7_* (repeat-> weekOfMonth - 1);
}

*priorDate = *nDate;
return SUCCESS;
}

/*-----
* Name:      goBackward
* Purpose:   for MONTHLY/YEARLY by POS AND CLASS,
*            adjust date from Last of month to desired day in month
* Author:    David Boothby, Copyright (c) IntelliLink, 1994
*-----*/
static void goBackward ( ILTR_PREPEAT repeat,
                        long IL_DIST *nDate, int nDay )
{
    /*-----
    * Tables of "subtracters" to get from date of Last day of month
    * to date of last weekday / weekend day of month
    *-----*/

    static int weekday_subtractor[7] =
    { // If Last day of the month is...
      //----- Sun Mon Tue Wed Thu Fri Sat, // then subtract number in table
        2,  0,  0,  0,  0,  0,  1 // to reach last weekday
    };
    static int weekend_subtractor[7] =
    { // If Last day of the month is...
      //----- Sun Mon Tue Wed Thu Fri Sat, // then subtract number in table
        0,  1,  2,  3,  4,  5,  0 // to reach last weekend day
    };

    switch (repeat->dayOfWeek)
    {
        case ILTR_ANY_DAY: // no adjustment necessary
            break;
        case ILTR_ANY_WEEKDAY:
            *nDate -= weekday_subtractor [nDay-1];
            break;
        case ILTR_ANY_WEEKEND_DAY:
            *nDate -= weekend_subtractor [nDay-1];
            break;
    }
}

/*-----
* Name:      goForward
* Purpose:   for MONTHLY/YEARLY by POS AND CLASS,
*            adjust date from First of month to desired day in month
* Author:    David Boothby, Copyright (c) IntelliLink, 1994
*-----*/
static void goForward ( ILTR_PREPEAT repeat,
                       long IL_DIST *nDate, int nDay )
{
    /*-----
    * Tables of "adders" to get from date of First day of month
    * to date of Nth weekday/weekendDay of the month.
    *-----*/

    static int weekday_adder[4][7] =
    { // If First day of the month is...
      //----- Sun Mon Tue Wed Thu Fri Sat, // then add number in table
        1,  0,  0,  0,  0,  0,  2, // to reach 1st weekday
        2,  1,  1,  1,  1,  3,  3, // to reach 2nd weekday
        3,  2,  2,  2,  4,  4,  4, // to reach 3rd weekday
        4,  3,  3,  5,  5,  5,  5 // to reach 4th weekday
    };
    static int weekend_adder[4][7] =

```

```

{ // If First day of the month is...
  //----- Sun Mon Tue Wed Thu Fri Sat, // then add number in table
    0,  5,  4,  3,  2,  1,  0,  // to reach 1st weekend day
    6,  6,  5,  4,  3,  2,  1,  // to reach 2nd weekend day
    7, 12, 11, 10,  9,  8,  7,  // to reach 3rd weekend day
    13, 13, 12, 11, 10,  9,  8  // to reach 4th weekend day
};
int stepsToTake = repeat->day - ILTR_POS_1ST;

switch (repeat->dayOfWeek)
{
  case ILTR_ANY_DAY:
    *nDate += stepsToTake;
    break;
  case ILTR_ANY_WEEKDAY:
    *nDate += weekday_adder [stepsToTake] [nDay-1] ;
    break;
  case ILTR_ANY_WEEKEND_DAY:
    *nDate += weekend_adder [stepsToTake] [nDay-1] ;
    break;
}
}

/*-----
* Name:      isLast
* Purpose: Check if date is last occurrence of day in given month
* Input:     Date
* Return:    TRUE if date is last occurrence, FALSE otherwise
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/
static int isLast (long nDate)
{
  int nDay;                // Day of week
  int month, day, year;    // Date components
  int holdDays;           // Temporary number of days
  int holdDay;            // Temporary day of week
  long holdDate;          // Temporary date

  //----- Compute day of week and total number of days in month.
  IL_DateDecode (nDate, &month, &day, &year);
  holdDays = IL_DaysInMonth (month, year);
  nDay = IL_DayOfWeek (nDate);

  //----- Compute date of last occurrence of given day of week.
  IL_DateEncode (month, holdDays, year, &holdDate);
  holdDay = IL_DayOfWeek (holdDate);
  holdDate = (holdDay < nDay) ?
    (holdDate - 7 + (nDay - holdDay)) :
    (holdDate - (holdDay - nDay));

  //----- Is it the last one?
  return (holdDate == nDate ? TRUE : FALSE);
}

/*-----
* Name:      normalizeDate
* Purpose: Normalize date fields after increment operation
* Input:     Pointers to year, month, week, dayOfWeek
* Return:    None
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/
static void normalizeDate ( int IL_DIST *year,
                          int IL_DIST *month,
                          int IL_DIST *week,
                          int IL_DIST *dow )
{
  //----- Normalize day of week and adjust week if necessary.
  if (dow != NULL && *dow > 7)
  {
    if (week != NULL)
      *week += ((*dow - 1) / 7);
    *dow %= 7;
    *dow = *dow ? *dow : 7;
  }
}

```

```

    }

    //----- Normalize week of month and adjust month if necessary.
    if (week != NULL && *week > 5)
    {
        if (month != NULL)
            *month += ((*week - 1) / 5);
        *week %= 5;
        *week = *week ? *week : 5;
    }

    //----- Normalize month and adjust year if necessary.
    if (month != NULL && *month > 12)
    {
        if (year != NULL)
            *year += ((*month - 1) / 12);
        *month %= 12;
        *month = *month ? *month : 12;
    }
}

/*-----
* Name:      normalizeDay
* Purpose:   Normalize day field to max for specified month & year
* Input:     Values of day, month & year
* Return:    Correct day for specified month/year
* Author:    Bill Berthoud Copyright (c) IntelliLink, 1993
*-----*/
static int normalizeDay ( int day,
                        int month,
                        int year)
{
    int    maxDays[12] = { 31, 28, 31, 30, 31, 30,
                          31, 31, 30, 31, 30, 31 };

    //----- Account for leap year
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
        maxDays[1] = 29;

    return (min (day, maxDays[month - 1]));
}

```

```

/*-----
* Name:      unfold.c
* Part of:   IntelliLink translation harness (ILTR library)
* Contents:  core logic for FANNING a repeating item into N simple instances
*
* Functions: ILItemHasInstancesInDateRange
*            ILTRUnfoldRepeat      -- higher-level entrypoint
*            ILUnfoldRepeat       -- lower-level entrypoint
*            AdjustDateRange
*            ComputeEnoughDays
*            Unfold
*            DiscardExcess
*            SetMaxFanout
*
* Author:    Copyright (c) IntelliLink, 1994-1996
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*        a "base" DLL rather than being statically linked into every
*        translator. Please ensure that all non-static functions in this
*        module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#include "iltr.h"

static int AdjustDateRange
( ILTR_PREPEAT pRepeat,
  int nMaxItems,
  long Today,
  long IL_DIST *pStartDate,    // IN/OUT lower bound
  long IL_DIST *pStopDate );  // IN/OUT upper bound

static int ComputeEnoughDays
( ILTR_PREPEAT pRepeat,
  long ItemCount,
  long IL_DIST *pDayCount );

static int Unfold
( ILTR_PREPEAT pRepeat,
  int nMaxItems,
  long StartDate,
  long StopDate,
  ILUT_PBUFFER pFanBuf,
  int IL_DIST *pGoodCount,    // OUT: # of unexcluded instances generated
  int IL_DIST *pGrossCount ); // OUT: # of instances (excluded+unexcluded)

static int DiscardExcess
( int nMaxItems,
  long Today,
  ILUT_PBUFFER pFanBuf,
  int IL_DIST *pGoodCount,    // IN/OUT: # of unexcluded instances
  int IL_DIST *pGrossCount ); // IN/OUT: # of instances

static int SetMaxFanout
( ILTR_PTRANSL tr,
  ILTR_PREPEAT pRepeat,
  ILTR_PFANOUT_MAXIMA pMaxima, // IN: V26 fanout limits
  int IL_DIST *pnMaxItems );   // OUT: computed limit for this pattern

/*-----
* ILItemHasInstancesInDateRange
* Callers:  EXPORT.C and IMPORT.C and TIFPUT.CPP -- DateRange checking
*
* Returns TRUE or FALSE or Error Code for Abnormal Error
*-----*/
int ILItemHasInstancesInDateRange          // Check for instances in range
( ILTR_PTRANSL tr,
  long lStartRange,                       // lower limit of date range
  long lEndRange,                         // upper limit of date range
  ILTR_PREPEAT pRepeat )                 // Repeat structure (supplied)
{
  int rc;
  int GoodCount;

```

```

int GrossCount;

/*-----
 * Fan out just ONE instance of the pattern into a simple Date Array
 * containing both excluded and un-excluded dates, in chronological order.
 * Excluded dates are NEGATIVE in the Date Array.
 *-----*/
rc = ILTRUnfoldRepeat ( tr, pRepeat,
                        1, // max fanout
                        ILTR_nDate, // "Today"
                        lStartRange, // Low end of fanning range
                        lEndRange, // High end of fanning range
                        NULL, // no maxima struct; we just want 1
                        &ILTR_pTmpBuf, // buffer into which to fan
                        &GoodCount, // count of unexcluded dates
                        &GrossCount ); // count of ALL dates

if (rc != SUCCESS)
    return ILERROR (rc, rc); // abnormal error

else
    return (GoodCount > 0);

} //---- ILItemHasInstancesInDateRange

/*-----
 * Name: ILTRUnfoldRepeat
 *
 * Callers: ILTR\repeat.c\ILRepeatItem
 *          ILTIF2\tifsync.cpp\TIFSyncFanOutRecurrencePattern
 *          ILXNOW\evrepeat.c\EventDBFanEvent
 *
 * This is the highest-level function in this module. It encapsulates the
 * Date Range and Max Count setup logic, plus the call to the
 * lower-level ILUnfoldRepeat function.
 *
 * NOTE: pass in -1 for lFanningMinDate to have Date Range established
 * by this function rather than setting it up yourself.
 *-----*/
ILBASEFN_int ILTRUnfoldRepeat
( ILTR_PTRANSL tr,
  ILTR_PREPEAT pRepeat, // IN: pattern to unfold
  int nMaxItems, // IN: pre-V26 fanout limits
  long lToday, // IN: today's date
  long lFanningMinDate, // IN: lower bound of Date Range
  long lFanningMaxDate, // IN: upper bound of Date Range
  ILTR_PFANOUT_MAXIMA pMaxima, // IN: V26 fanout limits
  ILUT_PBUFFER_IL_DIST *ppFanBuf, // IN/OUT: **buffer
  int IL_DIST *pGoodCount, // IN/OUT: # of unexcluded instances
  int IL_DIST *pGrossCount ) // IN/OUT: # of instances
{
    int rc;

    /*-----
     * If we have a 'maxima' structure to refer to, and if running under a
     * new enough engine, IGNORE and OVERWRITE the value in nMaxItems.
     *-----*/
    if (pMaxima && ILTR_VERSION_IS_AT_LEAST(26))
    {
        rc = SetMaxFanout (tr, pRepeat, pMaxima, &nMaxItems);
        if (rc != SUCCESS)
            return rc;
    }

    /*-----
     * If not supplied by caller, set Date Range limits for FANNING.
     * Starting with version 25 of ILTR we keep the fanning limits
     * separate from general date range limits, because TIF has to
     * fiddle with the general limits sometimes.
     *-----*/
    if (lFanningMinDate == -1)
    {
        if (ILTR_VERSION_IS_AT_LEAST(25))
        {
            lFanningMinDate = ILTR_lFanningMinDate;

```

```

        lFanningMaxDate = ILTR_lFanningMaxDate;
    }
    else
    {
        lFanningMinDate = ILTR_lDateRangeStart;    // tr->nLoDate
        lFanningMaxDate = ILTR_lDateRangeEnd;      // tr->nHiDate
    }
}

//---- Enforce the FUTURE option too
if ( (ILTR_nSchOpt == ILTR_RANGE_FUTURE)
    && (lFanningMinDate < lToday) )
    lFanningMinDate = lToday;

//---- Create reusable fanning buffer, if not already created
//---- NOTE: the caller's "world" must free this buffer eventually
if (*ppFanBuf == NULL)
{
    IL_HANDLE h;

    *ppFanBuf = (ILUT_PBUFFER) IL_ALLOC (sizeof (ILUT_BUFFER), h);
    if (*ppFanBuf == NULL)
        return ILTR_ERR_NOMEM;

    (*ppFanBuf)->hBufferHeader = h;

    rc = ILUT_InitBuffer (*ppFanBuf, 0, 512, 20000);
    if (rc != SUCCESS)
    {
        IL_FREE_AND_ZERO (h, *ppFanBuf)
        return ILERROR (rc, ILTR_ERR_NOMEM);
    }
}

/*-----
 * Fan out the pattern into a simple Date Array containing both
 * excluded and un-excluded dates, in chronological order.
 * Excluded dates are NEGATIVE in the Date Array.
 *-----*/
rc = ILUnfoldRepeat ( pRepeat, nMaxItems, lToday,
                    lFanningMinDate, lFanningMaxDate,
                    *ppFanBuf, pGoodCount, pGrossCount);

return rc;
} //---- ILTRUnfoldRepeat

/*-----
 * Name:      ILUnfoldRepeat
 *
 * This function unfolds a repeat pattern into an array of instance
 * dates. The array is constructed in a reusable buffer, managed
 * cooperatively by this function and its callers.
 *
 * This function returns SUCCESS unless something goes wrong.
 *
 * This "unfold" function is designed to produce good results for all cases,
 * including the following example:
 *
 * Suppose user has Date Range set to go from 1 year in the past up to
 * 2 years in the future, and suppose that the fanning limit for daily
 * appointments is 31. Then suppose he has a daily appointment called
 * "Lunch" that runs from Jan 1, 1900 through Dec 31, 2049. IntelliLink's
 * previous fanning logic would create 31 "Lunch" instances starting a
 * year ago ... hence there will be NONE in the present or future.
 *
 * The "unfold" algorithm puts as many fanned instances in the FUTURE as
 * possible (starting with TODAY). For example, if today is 3/31/96, and
 * we're fanning a YEARLY April Fools item that runs from 1900 through 1999,
 * with a fanout maximum of 10, we cannot fit all 10 instances in the future,
 * so we end up generating instances for the ten years 1990-1999. But if the
 * recurrence pattern had no STOP date, we would generate instances for the
 * ten years 1996-2005.
 *
 * If the current Date Range ends in the past, or starts in the future,

```

```

* the fanning logic will generate instances that fall as close to TODAY
* as possible.
*
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1996
*-----*/
ILBASEFN int ILUnfoldRepeat
( ILTR_PREPEAT pRepeat,      // IN: pattern to unfold
  int nMaxItems,             // IN: not to exceed this count (really!)
  long Today,                // IN: Today's date
  long lDateRangeStart,      // IN: date range lower bound
  long lDateRangeEnd,        // IN: date range upper bound
  ILUT_PBUFFER pFanBuf,      // IN/OUT: ptr to header for reusable buffer
  int IL_DIST *pGoodCount,    // OUT: # of unexcluded instances generated
  int IL_DIST *pGrossCount ) // OUT: # of instances (excluded+unexcluded)
{
    int rc;
    long StartDate = lDateRangeStart;
    long StopDate = lDateRangeEnd;

    //---- Make sure we've been asked to generate 1 or more instances
    if (nMaxItems < 1)
        return ILERROR (nMaxItems, ILTR_ERR_BAD_FANOUT_MAX);

    //---- adjust date range "inward" to favor most useful instances
    //---- we never adjust either bound "outward"
    rc = AdjustDateRange (pRepeat, nMaxItems, Today, &StartDate, &StopDate);
    if (rc != SUCCESS)
        return rc;

    /*-----
    * NOTE: there's a funny boundary condition for patterns with numDays > 1.
    * e.g. a weekly MONDAY appointment with numDays=3,
    *       with startDate = TUESDAY June 1st will have its first occurrence
    *       on MONDAY June 7th.
    *
    * Arguably this is correct, but also arguably the first occurrence should
    * should be TUESDAY June 2nd. There used to be code here to adjust the
    * pattern start date backward to try to find the earliest possible first
    * occurrence date for such cases. But that code introduced a bug.
    * Furthermore I now think it makes more sense to start a multi-day appt
    * of this sort on the designated day (Monday in this case) rather than
    * starting it in the midst of a "numDays" streak.
    *
    * NOTE: the startDate mentioned here is the PATTERN startDate, not the
    * DateRange start date. When the DateRange start date falls in the middle
    * of a numDays streak we guarantee that the earliest possible mid-streak
    * occurrence date is found.
    *-----*/

    //---- Generate ALL instances that fall within the adjusted bounds
    //---- NOTE: we typically generate MORE than 'nMaxItems' here
    rc = Unfold ( pRepeat, nMaxItems, StartDate, StopDate, pFanBuf,
                  pGoodCount, pGrossCount );

    //---- complain if the Unfold didn't work
    if (rc != SUCCESS)
        return rc;

    //---- if we generated too many instances, pare down the list & counts
    if (*pGoodCount > nMaxItems)
        rc = DiscardExcess (nMaxItems, Today, pFanBuf, pGoodCount, pGrossCount);

    return rc;
} //---- ILUnfoldRepeat

/*-----
* AdjustDateRange -- only caller is ILUnfoldRepeat
*
* Adjust date range that is used to bound the unfolding process, so that
* we produce approximately the right number of instances, erring on the
* side of generosity, and aiming to produce the most "useful" instances.
*
* Always adjust "inward" -- the lower bound may be raised, and the upper

```



```

* bound may be lowered.
*-----*/
static int AdjustDateRange
( ILTR_PREPEAT pRepeat,
  int nMaxItems,
  long Today,
  long IL_DIST *pStartDate,    // IN/OUT lower bound
  long IL_DIST *pStopDate )   // IN/OUT upper bound
{
    int rc;
    long ItemsDesired;
    long EnoughDays;
    long DaysInRange;

    //---- adjust inward if pattern itself is more tightly constrained
    //---- adjust start date upward if pattern starts later
    if (*pStartDate < pRepeat->startDate)
        *pStartDate = pRepeat->startDate;

    //---- adjust stop date downward if pattern stops sooner
    if ( (pRepeat->stopDate != -1)
        && (*pStopDate == 0 || *pStopDate > pRepeat->stopDate) )
        *pStopDate = pRepeat->stopDate;

    //---- no further adjustments needed for NOREPEAT items
    if (pRepeat->type == ILTR_NOREPEAT)
        return SUCCESS;

    //---- apply pragmatic constraints if DateRange has no sensible bounds
    if (*pStartDate == 0)
        IL_DateEncode(1, 1, 1950, pStartDate);    // Jan 1, 1950

    if (*pStopDate == 0)
        IL_DateEncode(12, 31, 2049, pStopDate);    // Dec 31, 2049

    //---- figure gross count wanted (unexcluded + excluded + fudge factor)
    ItemsDesired = (long) nMaxItems + (long) pRepeat->numExDates + 10L;

    /*-----
    * Figure out how many days we would need to traverse to
    * generate that many instances.
    *-----*/
    rc = ComputeEnoughDays (pRepeat, ItemsDesired, &EnoughDays);
    if (rc != SUCCESS)
        return rc;

    //---- how big a range do we actually have
    DaysInRange = (*pStopDate - *pStartDate) + 1;

    //---- if current range is "none too wide", we have nothing more to do here
    if (DaysInRange <= EnoughDays)
        return SUCCESS;

    //---- but if range is too wide, figure out how to tighten it up.
    //---- if whole range is in the future, just chop off the high end
    if (*pStartDate >= Today)
        *pStopDate = *pStartDate + EnoughDays - 1;

    //---- if whole range is in the past, just chop off the low end
    else if (*pStopDate <= Today)
        *pStartDate = *pStopDate - (EnoughDays - 1);

    //---- if range straddles the present, may have to chop both ends
    else
    {
        long FutureDaysInRange = (*pStopDate - Today) + 1;    // including today
        long PastDaysNeeded = EnoughDays - FutureDaysInRange;

        //---- if we can put the entire range in the future, let it start TODAY
        if (PastDaysNeeded <= 0)
        {
            *pStartDate = Today;
            *pStopDate = Today + EnoughDays - 1;
        }
        else
    }
}

```

```

//---- adjusted range will straddle the present
{
    long IdealStartDate = Today - PastDaysNeeded;

    //---- chop some off the low end if there's any to spare
    if (*pStartDate < IdealStartDate)
        *pStartDate = IdealStartDate;

    /*-----
    * And chop some off the high end if necessary. Barring logic errors
    * we know that this cannot commit the criminal act of EXTENDING the
    * stop date upward, because we verified way back when that the range
    * we're chopping is bigger than the range we want.
    *-----*/
    *pStopDate = *pStartDate + EnoughDays - 1;
}

return SUCCESS;
} //---- AdjustDateRange

/*-----
* ComputeEnoughDays -- only caller is AdjustDateRange
*
* Figure out how many days we would need to traverse to generate the
* desired ItemCount. Err on the generous side.
*-----*/
static int ComputeEnoughDays ( ILTR_REPEAT pRepeat,
                              long ItemCount,
                              long IL_DIST *pDayCount )
{
    long NumDays = (long) pRepeat->numDays;
    long Frequency = (long) pRepeat->frequency;
    long UnitSize; // days per unit (unit is week or month or ...)
    long Months = 1; // varies only for MIXED patterns
    long nMinDays = 0; // For certain patterns, want a minimum result
    int i;

    switch (pRepeat->type)
    {
        case ILTR_DAILY:
            UnitSize = 1; // DayCount = (ItemCount * Frequency * 1) / NumDays;
            break;

        case ILTR_WEEKLY:
            UnitSize = 7; // DayCount = (ItemCount * Frequency * 7) / NumDays;
            break;

        case ILTR_MONTHLY_BY_POS_AND_CLASS:
        case ILTR_MONTHLY_BY_POS:
        case ILTR_MONTHLY:
            UnitSize = 31; // DayCount = (ItemCount * Frequency * 31) / NumDays;
            break;

        case ILTR_QUARTERLY:
        case ILTR_QUARTERLY_BY_POS:
            UnitSize = 92; // DayCount = (ItemCount * Frequency * 92) / NumDays;
            break;

        case ILTR_YEARLY_BY_POS_AND_CLASS:
        case ILTR_YEARLY_BY_POS:
        case ILTR_YEARLY:
            UnitSize = 366; // DayCount = (ItemCount*Frequency * 366) / NumDays;
            break;

        case ILTR_WEEKLY_DAYS:
            UnitSize = 7;
            //---- Compute number of days selected per week
            NumDays = 0;
            for (i=0; i < 7; i++)
                if (pRepeat->days[i]) NumDays++;
    }
}

```

```

        break; // DayCount = (ItemCount * Frequency * 7) / NumDays;

    case ILTR_MIXED_BY_POS:
    case ILTR_MIXED:
        Frequency = 1; // no other frequencies allowed for mixed patterns
        UnitSize = 366;
        Months = 0;
        //----- Since MIXED is non-regular, provide a minimum of 1 year
        nMinDays = 366;

        for (i=0; i < 12; i++)
            if (pRepeat->months[i]) Months++;

        if (pRepeat->type == ILTR_MIXED)
            NumDays = 1; // DayCount = (ItemCount * 366) / Months;
        else
        {
            //---- For MIXED_BY_POS count selected weeks
            long NumWeeks = (long) ( pRepeat->weeks[0]
                                     + pRepeat->weeks[1]
                                     + pRepeat->weeks[2] );
            //---- Be conservative; count 4th & LAST weeks as a single week
            NumWeeks += (long) (pRepeat->weeks[3] | pRepeat->weeks[4]);

            //---- count days selected per week
            NumDays = 0;
            for (i=0; i < 7; i++)
                if (pRepeat->days[i]) NumDays++;

            //---- finally compute days per selected month
            NumDays *= NumWeeks;

            // DayCount = (ItemCount * 366) / (NumDays * Months);
        }

        break;

    case ILTR_NOREPEAT:
    default:
        return ILERROR ((int) pRepeat->type, ILTR_ERR_INTERNAL);
}

if (NumDays == 0 || Months == 0)
    return ILERROR (0, ILTR_ERR_REPEAT);

*pDayCount = (ItemCount * Frequency * UnitSize) / (NumDays * Months);

//----- Make sure the required minimum is returned
*pDayCount = max (*pDayCount, nMinDays);

return SUCCESS;
} //----- ComputeEnoughDays

/*-----
 * Unfold -- only caller is ILUnfoldRepeat
 *
 * Generate ALL instances that fall within the adjusted bounds.
 * Pay no attention to the start & stop dates in the ILTR_REPEAT structure;
 * use the StartDate and StopDate parameters passed in to this function.
 *-----*/
static int Unfold
(
    ILTR_REPEAT pRepeat,
    int nMaxItems,
    long StartDate,
    long StopDate,
    ILUT_PBUFFER pFanBuf,
    int IL_DIST *pGoodCount, // OUT: # of unexcluded instances generated
    int IL_DIST *pGrossCount // OUT: # of instances (excluded+unexcluded)
)
{
    int rc;
    int i, j;
    long IL_DIST *DateArray;
    int FullCount = 0;

```

```

int UnexcludedCount = 0;
long lPriorDate = -1;
long lDate = pRepeat->startDate;

//---- we intend for the limiting factor to be the Date Bounds, so set
//---- max count generously high.
int GrossLimit = nMaxItems + (int) pRepeat->numExDates + 200;
rc = ILUT_GetBuffer (pFanBuf, (long) (GrossLimit * sizeof(long)));
if (rc != SUCCESS)
    return ILERROR (rc, ILTR_ERR_NOMEM);

DateArray = (long IL_DIST *) pFanBuf->pBuffer;

/*-----
 * Fill up Date Array by unfolding the repeat pattern into it.
 * Count all instances generated by pattern, and keep a separate
 * count of all un-excluded instances.
 *-----*/
while (FullCount < GrossLimit)
{
    rc = ILRepeatNextDate (pRepeat, &lDate, &lPriorDate);
    if (rc != SUCCESS)
        return ILERROR(rc, ILTR_ERR_INTERNAL);

    //---- Iterate for the specified number of consecutive days.
    for (i = 0; i < pRepeat->numDays; i++, lDate++)
    {
        if (lDate < StartDate)
            continue;

        //---- stop fanning if stop date has been exceeded.
        if (lDate > StopDate)
            goto FINISHED_FANNING;

        //---- store and count new date -- assume it's not excluded
        DateArray [FullCount++] = lDate;
        UnexcludedCount++;

        //---- Look for this date in the exclusion list.
        for (j = 0; j < pRepeat->numExDates; j++)
        {
            if (lDate == pRepeat->exDates[j])
            {
                //---- date is excluded -- negate date & adjust count
                DateArray [FullCount-1] = -lDate;
                UnexcludedCount--;
                break;
            }
        }

        if (FullCount >= GrossLimit)
            goto FINISHED_FANNING;
    }

    //---- don't do it again if pattern is NOREPEAT
    if (pRepeat->type == ILTR_NOREPEAT)
        break;
} //---- while (FullCount < GrossLimit)

FINISHED_FANNING:
//-----

//---- pass back counts
*pGrossCount = FullCount;
*pGoodCount = UnexcludedCount;

return SUCCESS;
} //---- Unfold

/*-----
 * DiscardExcess -- only caller is ILUnfoldRepeat

```

```

*
* This function is only called when the 'Unfold' routine generates more
* un-excluded instances than we want. This function decides which ones
* to discard (either from low end or high end), does the DISCARD operation,
* and adjusts the counts accordingly.
*-----*/
static int DiscardExcess
( int nMaxItems,
  long Today,
  ILUT_PBUFFER pFanBuf,
  int IL_DIST *pGoodCount, // IN/OUT: # of unexcluded instances
  int IL_DIST *pGrossCount ) // IN/OUT: # of instances
{
    int Excess = *pGoodCount - nMaxItems;
    long IL_DIST *DateArray = (long IL_DIST *) pFanBuf->pBuffer;
    int GoodItems = 0;
    int NewFloor;
    int i;

    //---- count un-excluded items that are in the past
    //---- but don't count more than we need to...
    for (i=0; i < *pGrossCount; i++)
    {
        //---- once we reach TODAY, look no further
        if (DateArray[i] >= Today)
            break;

        else if (DateArray[i] > 0)
        {
            //---- make NewFloor point to item after the one we count here
            NewFloor = i+1;
            GoodItems++;
            if (GoodItems == Excess)
                break;
        }
    }

    //---- if we found any low-end items to chop off, shuffle & chop from below
    if (GoodItems > 0)
    {
        int Ceiling = *pGrossCount - NewFloor;
        int GoodItemsShuffled = 0;

        for (i=0 ;i < Ceiling; i++)
        {
            DateArray[i] = DateArray[NewFloor+i];
            if (DateArray[i] > 0)
            {
                GoodItemsShuffled++;
                if (GoodItemsShuffled == nMaxItems)
                {
                    //---- we have shuffled enough items. we're done!!
                    *pGoodCount = nMaxItems;
                    *pGrossCount = i+1; // Date Array [0..i]
                    return SUCCESS;
                }
            }
        }

        return ILERROR (i, ILTR_ERR_INTERNAL); // should never get here
    } //---- if (GoodItems > 0)

    //---- nothing chopped off low end. All chopping must be from high end.
    GoodItems = 0;
    for (i = *pGrossCount-1; i >= 0; i--)
    {
        if (DateArray[i] > 0)
        {
            //---- count good items that we chop, until we've chopped enough
            GoodItems++;
            if (GoodItems == Excess)
            {
                *pGoodCount = nMaxItems;
            }
        }
    }
}

```

```

        *pGrossCount = i;          // Date Array [0..i-1]
        return SUCCESS;
    }
}

return ILERROR-(i, ILTR_ERR_INTERNAL); // should never get here
} //----- DiscardExcess

/*-----
 * SetMaxFanout -- called by ILTRUnfoldRepeat
 *
 * Pick "base" max factor from the maxima structure passed in, then
 * multiply by NumDays if the max factor is NEGATIVE.
 *-----*/
static int SetMaxFanout
( ILTR_PTRANSL tr,
  ILTR_PREPEAT pRepeat,
  ILTR_PFANOUT_MAXIMA pMaxima, // IN: V26 fanout limits
  int IL_DIST *pnMaxItems )    // OUT: computed limit for this pattern
{
    int max;
    int i;
    int NumDays = (int) pRepeat->numDays;

    switch (pRepeat->type)
    {
        case ILTR_DAILY:          max = pMaxima->nDailyMaxFanout;      break;

        case ILTR_WEEKLY_DAYS:
            //----- Compute number of days selected per week
            NumDays = 0;
            for (i=0; i < 7; i++)
                if (pRepeat->days[i]) NumDays++;

            //----- fall through to next case...

        case ILTR_WEEKLY:          max = pMaxima->nWeeklyMaxFanout;      break;

        case ILTR_MONTHLY_BY_POS_AND_CLASS:
        case ILTR_MONTHLY_BY_POS:
        case ILTR_MONTHLY:          max = pMaxima->nMonthlyMaxFanout;      break;

        case ILTR_QUARTERLY_BY_POS:
        case ILTR_QUARTERLY:          max = pMaxima->nQuarterlyMaxFanout; break;

        case ILTR_YEARLY_BY_POS_AND_CLASS:
        case ILTR_YEARLY_BY_POS:
        case ILTR_YEARLY:          max = pMaxima->nYearlyMaxFanout;      break;

        case ILTR_MIXED_BY_POS: //----- we could get really fancy here!!
        case ILTR_MIXED:
        case ILTR_NOREPEAT:
        default:                    max = pMaxima->nOtherMaxFanout;      break;
    }

    if (max < 0)
        max *= -NumDays;

    *pnMaxItems = max;
    return SUCCESS;
} //----- SetMaxFanout

```

```

/*-----
* Name:      ILGetRepeat
* Purpose: Retrieve repeating item from intermediate file
* Input:     Pointers to translation and repeating structures
* Return:    SUCCESS or error code
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/

/*-----
* NOTE:  this is an "ILBASE" module, which means that it may be built into
*        a "base" DLL rather than being statically linked into every
*        translator. Please ensure that all non-static functions in this
*        module are declared with ILBASE_xxxx macros, defined in ILTYPES.H.
*-----*/

#define __MSC
#include "iltr.h"

ILBASEFN_int ILGetRepeat ( ILTR_PTRANSL tr,
                          ILTR_PREPEAT repeat )
{
    int rc;                      // Return code
    int memSize;                 // Memory size
    long tempLen;                // Temporary field length
    char szRptBuf [sizeof(ILTR_REPEAT)+2];

    /*-----
    * PROLOGUE: Special rules apply to the format of repeating
    * items in the intermediate file. The repeating item fields
    * are hidden from the user and do not appear in the field map.
    * These fields have pre-specified names which are purely
    * internal to the engine. When a repeating item is placed
    * in the intermediate file, two hidden fields are automatically
    * attached to the intermediate record: one field holding the
    * full contents of the ILTR_REPEAT structure and another
    * holding any exclusion dates. Unlike data fields,
    * these "meta" fields are stored in binary format and retrieved
    * exclusively by this routine.
    *-----*/

    //----- Clear repeating structure.
    IL_MEMSET (repeat, '\0', sizeof (ILTR_REPEAT));

    /*-----
    * Retrieve the BASIC repeating field data to determine if this is a
    * repeating item.
    *-----*/
    IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);

    /*-----
    * OK, this is gross. If the source translator uses long pointers
    * and the target uses short, provide a buffer big enough to hold
    * the full repeat field, which contains a pointer value. Then, copy
    * only the information we are interested in into the actual repeat
    * structure. The ILTR_REPEAT structure contains a dummy trailing
    * value which provides the necessary slack for the converse case.
    *-----*/
    tempLen = sizeof (szRptBuf);
    rc = ILTRGetField (tr, ILTR_REP_BASIC, szRptBuf, &tempLen);
    if (rc != SUCCESS)
        return ILTR_ERR_NOFLD;

    IL_MEMMOVE ( repeat, szRptBuf,
                (size_t) (min (tempLen, sizeof (ILTR_REPEAT) - 2)) );

    //----- Clear pointers and handles.
    repeat->exDates = NULL;
    repeat->hExDates = IL_NULL_HANDLE;

    //----- Single item with no repeating fields.
    if (tempLen == 0)
    {
        repeat->type = ILTR_NOREPEAT;
        repeat->frequency = 1;
        repeat->numDays = 1;
    }
}

```

```
        return SUCCESS;
    }

    /*-----
    * Are there any exclusion dates with this repeating item?
    * If so, allocate dynamic memory to store them.
    *-----*/
    if (repeat->numExDates == 0)
        return SUCCESS;          //----- No exclusion dates.
    else
    {
        //----- Allocate memory for entire exclusion list.
        memSize = sizeof (long) * repeat->numExDates;
        IL_ALLOC_MEM (memSize, repeat->hExDates, repeat->exDates);

        //----- Memory allocation error.
        if (repeat->exDates == NULL)
            return ILTR_ERR_NOMEM;

        //----- Retrieve all exclusion dates from intermediate file.
        IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);
        tempLen = repeat->numExDates * sizeof (long);
        rc = ILTRGetField (tr, ILTR_REP_XDATE, repeat->exDates, &tempLen);
        if (rc != SUCCESS)
            return ILTR_ERR_NOFLD;

        //----- Return indication that we found a repeating item.
        return SUCCESS;
    }
}
```



```

#include "iltr.h"

/*-----
 * Name:      ILPutRepeat
 * Purpose:   Place repeating item into intermediate file
 * Input:     Pointers to structures, start field name, stop field name
 * Return:    SUCCESS or error code
 * Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992
 *-----*/
int IL_DECL ILPutRepeat ( ILTR_PTRANSI tr,
                          ILTR_PREPEAT repeat,
                          IL_PSTR startField,
                          IL_PSTR stopField )
{
    int rc;                                // Return code

    /*-----
     * PROLOGUE: Special rules apply to the format of repeating
     * items in the intermediate file. The repeating item fields
     * are hidden from the user and do not appear in the field map.
     * These fields have pre-specified names which are purely
     * internal to the engine. When a repeating item is placed
     * in the intermediate file, two hidden fields are automatically
     * attached to the intermediate record: one field holding the
     * full contents of the ILTR_REPEAT structure and another
     * holding any exclusion dates. Unlike data fields,
     * these "meta" fields are stored in binary format and retrieved
     * exclusively by routine ILGetRepeat.
     *-----*/

    //----- Make sure that this is a legitimate repeating item.
    if (repeat->type == ILTR_NOREPEAT && repeat->numDays == 1)
        return SUCCESS;

    /*-----
     * Move start and stop date field names into REPEAT structure
     * before writing to intermediate file.
     *-----*/
    IL_STRCPY (repeat->startField, startField);
    IL_STRCPY (repeat->stopField, stopField);

    /*-----
     * Place the full contents of the REPEAT structure as a binary
     * field in the intermediate file (ILIF or TIF).
     * This is NOT a 'special fanning adjustment' situation.
     * Don't do character-mapping and don't feed data to filters mechanism.
     *-----*/
    rc = ILTRPutField ( tr, ILTR_REP_BASIC, (IL_PSTR) repeat,
                       (long) sizeof (ILTR_REPEAT), FALSE, FALSE, FALSE );
    if (rc != SUCCESS)
        return ILTR_ERR_NOFLD;

    //----- Write out any exclusion dates to intermediate file (ILIF or TIF).
    if (repeat->numExDates)
    {
        /*-----
         * Put the exclusion list into the intermediate file (ILIF or TIF).
         * This is NOT a 'special fanning adjustment' situation.
         * Don't do character-mapping and don't feed data to filters mechanism.
         *-----*/
        rc = ILTRPutField ( tr, ILTR_REP_XDATE, (IL_PSTR) repeat->exDates,
                           (long) repeat->numExDates * sizeof(long),
                           FALSE, FALSE, FALSE );
        if (rc != SUCCESS)
            return ILTR_ERR_NOFLD;
    }

    //----- Return without error.
    return SUCCESS;
}

```

```

/*-----
* Name:      EXPORT.C
* Part of:   IntelliLink Translation Harness (ILTR)
* Purpose:   Contains top-level "driver" entrypoint, ILEexport, for export
*            of record-oriented data.
*
* Functions:
*            ILEexport
*            ExportBegin
*            ExportInit1
*            ExportInit2
*            ExportWhile
*            ExportApplyFilters
*            ExportCheckApptDate
*            ExportApplyUserFilter
*            ExportAddLogEntry
*            ExportEnd
*            ExportRecord
*
* While Exporting under ILWIN.EXE, ILWIN\xlatew.c\LoadTranslator sets
* ILTR_pDriver to point to ILEexport. Then ILX_V3\dlgprog.c\ILX_DlgProgress
* calls (*ILTR_pDriver) repeatedly.
* (NOTE: Message loop is in ILWIN\xlatew.c\LoadTranslator.)
*
* While Exporting under Windows ILX, ILX_V3\xlate.c\doTranslate calls
* loadxltr.c\ILX_LoadTranslator, which sets ILTR_pDriver to point to ILEexport.
* Then ILX_V3\dlgprog.c\ILX_DlgProgress calls (*ILTR_pDriver) repeatedly.
* (NOTE: Message loop is in ILX_LoadTranslator.)
*
* While Exporting on Macintosh, ILX_V3\xlate.c\doTranslate
* calls ILX_V3\macdrive.cpp\ILX_LoadTranslator, which
* calls ILTR\macstub.c\CallCodeResource for function=ILX_CALL_EXPORT, which
* calls ILTR\macstub.c\RunXlator with pXlator=&ILEexport. Then RunXlator
* calls *pXlator repeatedly.
*
* NOTE:      this module is force-linked into every record-oriented translator
*
* Input:      Pointer to translator record
* Return:      SUCCESS, FAILURE, or error code
* Author:      Mike Blanchette, Copyright (c) IntelliLink, 1992
*-----*/

// If using MFC with MSVC 4.1 we need to call AfxGetInstanceHandle
#ifdef ILMFC41
    #include "afxwin.h"
#endif // ILMFC41

#include "ilxapi.h" // Includes ALL common headers

// for now WP stuff is only on the MAC
#ifdef ILMAC
    #include "iltrwp.h"
#endif

//----- Global variables
IL_HINST hXlatorInst; // Translator DLL instance

//----- local functions
static int ExportBegin (ILTR_PTRANSL tr);
static int ExportInit1 (ILTR_PTRANSL tr);
static int ExportInit2 (ILTR_PTRANSL tr);
static int ExportWhile (ILTR_PTRANSL tr);
static int ExportApplyFilters (ILTR_PTRANSL tr, int nRc);
static int ExportCheckApptDate (ILTR_PTRANSL tr, int nRc);
static int ExportApplyUserFilter (ILTR_PTRANSL tr, int nRc);
static int ExportAddLogEntry (ILTR_PTRANSL tr);
static int ExportEnd (ILTR_PTRANSL tr);
static int ExportRecord (ILTR_PTRANSL tr);

/*-----
* Windows DLL initiator function (DllMain or LibMain)
*-----*/

// If using MFC and MSVC 4.1 we can't use either of these
#ifdef ILMFC41

```

```

#ifdef ILWIN
    #ifdef WIN32

        //----- DLL initiator function for WIN32
        BOOL APIENTRY DllMain ( HANDLE hModule,
                                -          DWORD ul_reason_for_call,
                                LPVOID lpReserved )
        {
            //----- Capture the instance handle for later use
            if (ul_reason_for_call == DLL_PROCESS_ATTACH)
                hXlatorInst = (IL_HINST) hModule;
            return TRUE;
        }

    #else

        //----- DLL initiator function for WIN16
        int FAR PASCAL LibMain ( HANDLE hInstance,
                                WORD wDataSeg,
                                WORD wHeapSize,
                                LPSTR lpszCmdLine )
        {
            //----- Capture the instance handle for later use
            hXlatorInst = hInstance;
            if (wHeapSize > 0)
                UnlockData (0);
            return 1;
        }

    #endif // WIN32
#endif // ILWIN
#endif // ILMFC41

/*-----
 * Name:      ILEExport
 *-----*/
int IL_DECL EXP ILEExport (ILTR_PTRANSI tr)
{
    int nRc;

    // If no dllMain was called we need an instance handle
    #ifdef ILMFC41
        hXlatorInst = AfxGetInstanceHandle();
    #endif // ILMFC41

    /*-----
     * Word Processing sections use a different version of ILEExport.
     * If this is a WP translation, then call ILEExportWP.
     *-----*/
    #ifdef ILMAC
        if (ILTR_nFunction == ILTB_SEC_MEMO)
        {
            nRc = ILEExportWP (tr);
            return (nRc);
        }
    #endif

    switch (ILTR_nProcessStage)
    {
        //----- Initialize translator.
        case ILTR_BEGIN:

            nRc = ExportBegin (tr);
            if (nRc == SUCCESS)
                nRc = ILProcessMessages (tr);
            if (nRc == SUCCESS)
                ILTR_nProcessStage = ILTR_WHILE;
            return nRc;

        //----- Export a record
        case ILTR_WHILE:

            nRc = ExportWhile (tr);
            if (nRc == SUCCESS)
                nRc = ILProcessMessages (tr);
    }
}

```

```

        else if (nRc == ILTR_EOF)
        {
            ILTR_nProcessStage = ILTR_END;
            nRc = ILProcessMessages (tr);
        }
        return nRc;

//----- Shut down translator
case ILTR_END:

    nRc = ExportEnd (tr);
    if (nRc == SUCCESS)
        nRc = ILProcessMessages (tr);
    return nRc;

default:

    return ILERROR (ILTR_nProcessStage, ILTR_ERR_UNKNOWN);
}

} //---- IExport

/*-----
* Name:      ExportBegin (called from IExport)
*
* This function's behavior, in conjunction with the behavior of "dlgprog",
* implies the following basic cleanup rules for translator writers:
*
* (1) if your INIT returns an error, it must clean up any
*     allocated resources before doing so.
*
* (2) if your INIT returns SUCCESS, we guarantee that your END
*     routine will be called, so you can make your END routine
*     responsible for cleaning up any resources allocated by INIT.
*
* (3) But for historical reasons there is one more caveat:
*     If your BEGIN function returns ILTR_ERR_NODDE then before doing
*     so it must clean up any resources allocated by INIT.
*
*     When you return ILTR_ERR_NODDE, ILX_V3\dlgprog.c\ILX_DlgProgress
*     then tries to LAUNCH the source app, and if that succeeds then
*     your INIT and BEGIN functions will be called again.
*
* (4) Another peculiar case is ILTR_ERR_WAIT. But it is handled
*     differently. If your BEGIN return ILTR_ERR_WAIT then it will
*     be called again, but in this case your INIT will NOT be called
*     again.
*-----*/
static int ExportBegin (ILTR_PTRANSL tr)
{
    int nRc;

    //---- Do some initializations
    if (ILTR_bExpInitialized == FALSE)
    {
        nRc = ExportInit1 (tr);
        if (nRc != SUCCESS)
            return nRc;

        ILTR_bExpInitialized = TRUE;
        ILTR_bInitDone = FALSE;
        ILTR_bMustCleanUpBeforeQuitting = FALSE;
    }

    //---- Do app-specific initializations and some more initializations
    if (ILTR_bInitDone == FALSE)
    {
        //----- Call app-specific initialization function
        nRc = ILIniTranslator (tr, &ILTR_appData);
        if (nRc != SUCCESS)
            return nRc; // Possibly ILTR_ERR_WAIT

        /*-----
        * Beyond this point we must always do cleanup, whether job

```

```

    * runs to completion, or fails prematurely, or is cancelled by
    * the user. Note that ILX\DLGPROG.C checks the 'MustCleanUp' flag.
    *-----*/
    ILTR_bMustCleanUpBeforeQuitting = TRUE;

    //---- do some more initializations
    nRc = ExportInit2 (tr);
    if (nRc != SUCCESS)
        return nRc;

    /*-----
    * Set flag to denote that INIT is done.
    * But beware of NODDE special case below!
    *-----*/
    ILTR_bInitDone = TRUE;
}

if (ILTR_cbBegin != NULL)
{
    nRc = (*ILTR_cbBegin) (tr, &ILTR_appData);
    if (nRc != SUCCESS)
    {
        //---- Special case for NODDE: force INIT to be called again!!
        if (nRc == ILTR_ERR_NODDE)
            ILTR_bInitDone = FALSE;

        return nRc;
    }
}

//----- Set total record count in progress bar control.
ILStatusExport (tr, ILTR_nRecords);

//----- Write out record in log file.
if (ILTR_log)
{
    int resnum;
    if (ILTR_nSynchronize)
    {
        if (ILTR_phase == ILTR_PHASE10)
            resnum = ILTR_MSG_INPUT_1ST;
        else
            resnum = ILTR_MSG_INPUT_2ND;
    }
    else
    {
        if (ILTR_phase == ILTR_PHASE10)
            resnum = ILTR_MSG_REATAR;
        else
            resnum = ILTR_MSG_REASRC;
    }

    IL_SYNC_MEM (ILTR_hRes.hNdx, ILTR_hRes.pNdx);
    nRc = ILAppendLog (ILTR_hLog, ILTR_hRes, resnum, NULL, NULL);
    if (nRc != SUCCESS)
        return ILERROR (nRc, ILTR_ERR_LOGFILE);
}

//----- Get type of DATE field.
if (ILTR_nFunction == ILTR_APPT && ILTR_map.ApptDate != -1)
{
    unsigned long lAttrib;
    char szFldName[ILTR_MAX_FLDNAME];

    ILFldName (tr, ILTR_map.ApptDate, szFldName, ILTR_MAX_FLDNAME);
    ILFldType (tr, szFldName, &ILTR_cDateType, &lAttrib);
}

return SUCCESS;
} //---- ExportBegin

/*-----

```

```

* Name:      ExportInit1 (called from ExportBegin)
*-----*/
static int ExportInit1 (ILTR_PTRANSL tr)
{
    int nRc;
    char szMsgText[MAX_MSG];          // Display text

    //---- set signature for Heap Logging (if enabled)
    ILUT_MemSig (ILTR_szSrcTrans);

    //----- Signal start of import/export operation to progress window
    ILStatusBegin (tr);

    //----- Show Please Wait message.
    ILSTMakeString (&ILTR_hRes, ILTR_MSG_WAIT, szMsgText, MAX_MSG);
    ILSetProgressText (tr, szMsgText);

    //----- Clear variables and set default line terminator string.
    ILTR_lEarliestDate = 0L;
    ILTR_lLatestDate  = 0L;
    ILTR_recNum       = 0;
    IL_STRCPY (ILTR_szLineTerm, ILTR_DEFAULT_TERM);
    ILTR_appData = NULL;

    //----- Set ILTIME global default date and time format
    IL_InitStdDateFormat ();
    IL_InitStdTimeFormat ();

    //----- set ILTR date and time format
    IL_InitDateFormat ();
    IL_InitTimeFormat ();

    //----- When Schedule Range is FUTURE, set low date to today.
    if (ILTR_nFunction == ILTR_APPT && ILTR_nSchOpt == ILTR_RANGE_FUTURE)
        ILTR_lEarliestDate = ILTR_nDate;

#ifdef ILWIN
    //----- Make environment ID available as a window property
    SetProp (ILTR_hParentWin, ILTR_ENV_PROP,
            (IL_HANDLE) ILTR_eEnvironment);
#endif

    if (ILTR_pTmpBuf == NULL)
    {
        //---- initialize reusable buffer ILTR_pTmpBuf
        IL_ALLOC_MEM (sizeof (ILUT_BUFFER), ILTR_hTmpBuf, ILTR_pTmpBuf);
        if (ILTR_pTmpBuf == NULL)
            return ILTR_ERR_NOMEM;

        nRc = ILUT_InitBuffer (ILTR_pTmpBuf, 0, 512, ILTR_MAX_FIELDLLENGTH+1);
        if (nRc != SUCCESS)
            return ILTR_ERR_NOMEM;
    }

    //----- if doing an ILIF-based job, set up memory for ILIF to use...
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        //----- Allocate "play area" for ILIF to use
        IL_ALLOC_MEM (sizeof (ILIF_GLOBALS),
            ILTR_hILIF_Globals, ILTR_pILIF_Globals);
        if (ILTR_pILIF_Globals == NULL)
            return ILTR_ERR_NOMEM;

        //----- Zero play area
        IL_MEMSET (ILTR_pILIF_Globals, 0, sizeof (ILIF_GLOBALS));
    }

    return SUCCESS;
} //---- ExportInit1

/*-----
* Name:      ExportInit2 (called from ExportBegin)
*-----*/
static int ExportInit2 (ILTR_PTRANSL tr)

```

```

{
    int nRc;
    int numSections = 1;           // Number of sections in ILIF
    int numFields;                // Number of top-level fields
    char szSection[MAX_SECNAME_SIZE]; // Section name

    /*-----
    * Ensure that the GET callback routine has been registered.
    * Otherwise return with error indication.
    *-----*/
    if (ILTR_cbGet == NULL)
        return ILTR_ERR_NOGETCB;

    //----- Init all field and char mappings (must do before ILTIFReopenFile)
    nRc = ILSetupTables (tr);
    if (nRc)
        return nRc;

    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        //----- Initialize Intermediate File (ILIF) subsystem.
        nRc = ILIFInit ( ILIF_ACC_SINGLE_FILE,
                        ILTR_szWorkFile,
                        IL_ATTR_WRITE,
                        ILTR_NOT_USED,
                        (int IL_DIST *) &numSections );
        if (nRc != SUCCESS)
            //----- ERROR - could not initialize ILIF
            return ILERROR (nRc, ILTR_ERR_NOIF);
    }

    else
    {
        //----- ILX_V4: wake up TIF
        nRc = ILTR_ILTIFReopenFile (tr);
        if (nRc != SUCCESS)
            return nRc;
    }

    //----- Allocate memory for field buffer of default size.
    ILTR_field.width = MAX_FIELD_LEN;
    IL_ALLOC_MEM (ILTR_field.width,
                  ILTR_field.handle,
                  ILTR_field.buffer);
    if (ILTR_field.buffer == NULL)
        return ILTR_ERR_NOMEM;

    //----- Filters not currently supported on the MAC
    #ifndef ILMAC
        //----- Is a filter active?
        if (ILTR_nFilterID != -1)
        {
            //----- Load the filter.
            ILFilterStartup (tr);

            //----- Allocate memory for filter field buffer of default size
            ILTR_FilterFieldBuffer.width = MAX_FIELD_LEN;
            IL_ALLOC_MEM (ILTR_FilterFieldBuffer.width,
                          ILTR_FilterFieldBuffer.handle,
                          ILTR_FilterFieldBuffer.buffer);
            if (ILTR_FilterFieldBuffer.buffer == NULL)
                return ILTR_ERR_NOMEM;
        }
    #endif

    //----- Retrieve the number of top-level field names being exported.
    numFields = ILTR_list.Count;

    /*-----
    * Adjust the total number of fields to include the hidden
    * repeating fields if the function is APPT, TODO, or CALL.
    *-----*/
    if ( ILTR_nFunction == ILTR_APPT
        || ILTR_nFunction == ILTR_TODO
        || ILTR_nFunction == ILTR_CALL )
        numFields += ILTR_EXTRA_FIELDS_FOR_REPEAT;
}

```

```

/*-----
 * Adjust the total number of fields to include _appData
 * and _subType fields.
 *-----*/
numFields += ILTR_EXTRA_FIELDS_ALWAYS;

if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
{
    //----- Create intermediate file view.
    IL_SPRINTF (szSection, "%d", ILTR_nFunction);
    ILTR_view = ILIFCreate ( szSection,
                            numFields,
                            ILTR_CTRLA_CHAR,
                            ILTR_CTRLB_CHAR );
    if (ILTR_view == NULL_ILIF)
        //----- Error. Unable to create view.
        return ILTR_ERR_NOIF;

    //----- Create field descriptors in intermediate file.
    nRc = ILFldDesc (tr);
    if (nRc != SUCCESS)
        return nRc;
}

return SUCCESS;
} //----- ExportInit2

/*-----
 * Name:      ExportWhile (called from IExport)
 *-----*/
static int ExportWhile (ILTR_PTRANSL tr)
{
    int nRc, rc2;

    //----- Set default values for record name and action type.
    IL_MAKE_STRING NULL (ILTR_szRecName);
    ILTR_action = ILTR_ACT_READ;

    //----- Clear error fields.
    ILTR_nFldErrorNum = 0;
    IL_MEMSET (ILTR_fldError, '\0', sizeof (ILTR_FLDERR) * ILTR_MAX_FLDERR);

    /*-----
     * Now we get ready for fields to be accepted by ILIF or by TIF.
     * The tr structure member ILTR_phase tells us which to use.
     *-----*/
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);
        ILIFInitRecord (ILTR_view, ILTR_rec.buffer, ILTR_rec.width);

        /*-----
         * If running under an App that's new enough to use SST stuff,
         * and SST Tagging mechanism is enabled,
         * initialize record subtype to match source section subtype.
         * NOTE: for ILX_V4 mode this is done by ILTIF2\TIFInitRecord
         *-----*/
        if ( ILTR_VERSION_IS_AT_LEAST(21)
            && ((ILTR_Flags & ILTR_DISABLE_SST_TAGGING) == 0) )
        {
            char szTemp[20];

            IL_SPRINTF (szTemp, "%d", ILTR_SourceSST);
            nRc = ILFldPut (tr, ILTR_SUB_TYPE, szTemp, IL_STRLEN(szTemp));
            if (nRc != SUCCESS)
                return ILERROR(nRc, ILTR_ERR_INTERNAL_ERROR);
        }
    }
    else
    {
        nRc = ILTR_ILTIFInitRecord(tr);
        if (nRc != SUCCESS)
            return nRc;
    }
}

```



```

    }

    //----- Invoke callback function to export data record.
    nRc = (*ILTR_cbGet) (tr, &ILTR_appData);

    //----- End of file.
    if (nRc == ILTR_EOF)
    {
        ILTR_nProcessStage = ILTR_END;
        return SUCCESS;
    }
    else
        ILTR_recNum++; //----- count it: one more record has been read

    if (nRc > ILTR_SKIP_ALL || nRc < 0)
        //----- Bail out on serious error
        return nRc;

    nRc = ExportApplyFilters (tr, nRc);
    if (nRc > ILTR_SKIP_ALL || nRc < 0)
        //----- Bail out on serious error
        return nRc;

    /*-----
    * Has a value been supplied for record title? If not, attempt
    * to retrieve default value from "show" field specified in
    * field map table.
    *-----*/
    if (IL_STRING_IS_NULL(ILTR_szRecName))
    {
        //----- Is there a VIEW field defined in field list?
        if (ILTR_map.ShowSource != -1)
        {
            char szFldName[ILTR_MAX_FLDNAME]; // Field name
            long len;                        // Field length

            //----- Retrieve name and value of VIEW field.
            ILFldName (tr, ILTR_map.ShowSource, szFldName, ILTR_MAX_FLDNAME);
            len = (long) MAX_MSG;
            rc2 = ILTRGetField (tr, szFldName, ILTR_szRecName, &len);
        }
    }

    //----- Update progress bar.
    if ((nRc & ILTR_SKIP_METER) == 0)
        ILShowProgress (tr);

    //----- Don't count record if it was skipped.
    if (ILTR_action == ILTR_ACT_SKIP)
        ILTR_recNum--;

    //----- Add entry to logfile
    if (ILTR_log && ((nRc & ILTR_SKIP_LOG) == 0))
    {
        rc2 = ExportAddLogEntry (tr);
        if (rc2 != SUCCESS)
            return rc2;
    }

    if ((nRc & ILTR_SKIP_WRITE) == 0)
    {
        rc2 = ExportRecord (tr);
        if (rc2 != SUCCESS)
            return rc2;
    }

    return SUCCESS;
} //---- ExportWhile

/*-----
* Name:      ExportApplyFilters (called from ExportWhile)
*-----*/
static int ExportApplyFilters (ILTR_PTRANS tr, int nRc)
{

```

```

//---- Note that nRc is passed IN, then returned OUT

/*-----
 * Apply SST (Section SubType) filtering to any records
 * that we haven't already decided to skip.
 *-----*/
if ((ILTR_action == ILTR_ACT_READ) && ((nRc & ILTR_SKIP_WRITE) == 0))
{
    nRc = ILSST_Filter (tr);
    if (nRc > ILTR_SKIP_ALL || nRc < 0)
        return nRc;
}

/*-----
 * Apply Date Range analysis to any appointments
 * that we haven't already decided to skip.
 *-----*/
if ( (ILTR_nFunction == ILTR_APPT)
    && (ILTR_action == ILTR_ACT_READ)
    && ((nRc & ILTR_SKIP_WRITE) == 0) )
{
    nRc = ExportCheckApptDate (tr, nRc);
    if (nRc > ILTR_SKIP_ALL || nRc < 0)
        return nRc;
}

/*-----
 * Apply user-defined Filters to any records
 * that we haven't already decided to skip
 *-----*/
if ((ILTR_action == ILTR_ACT_READ) && ((nRc & ILTR_SKIP_WRITE) == 0))
{
    nRc = ExportApplyUserFilter (tr, nRc);
    if (nRc > ILTR_SKIP_ALL || nRc < 0)
        return nRc;
}

return nRc;
} //---- ExportApplyFilters

/*-----
 * Name:      ExportCheckApptDate (called from ExportApplyFilters)
 *-----*/
static int ExportCheckApptDate (ILTR_PTRANSI tr, int nRc)
{
    //---- Note that nRc is passed IN, then returned OUT

    long lStartDate = 0;                // Appointment start date
    BOOLEAN bRepeatAppt = FALSE;        // Is this a repeating appointment?
    BOOLEAN bSkipRecord = FALSE;        // Should record be skipped?
    ILTR_REPEAT rpt;
    int rc2;

    //----- Is this a repeating appointment? (check, and get repeat structure)
    bRepeatAppt = ILIsRepeat2 (tr, &rpt);
    if (bRepeatAppt)
    {
        //----- Ensure that dates are not past.
        if (ILTR_nSchOpt == ILTR_RANGE_FUTURE && rpt.startDate < ILTR_nDate)
            if (rpt.stopDate != -1 && rpt.stopDate < ILTR_nDate)
                bSkipRecord = TRUE;

        //----- Skip repeating items out of range
        if (ILTR_nLoDate && ILTR_nHiDate)
        {
            rc2 = ILItemHasInstancesInDateRange ( tr,                //---- see unfold.c
                                                  ILTR_nLoDate,
                                                  ILTR_nHiDate,
                                                  &rpt );

            //----- Free up exclusion list
            if (rpt.numExDates > 0)
            {
                IL_FREE_MEM_AND_ZERO_HANDLE (rpt.hExDates, rpt.exDates);
            }
        }
    }
}

```

```

        if (rc2 == FALSE)
            bSkipRecord = TRUE;
        else if (rc2 != TRUE)
            return rc2;          // abnormal error
    }
}

//----- Processing an individual appointment.
else if (ILTR_map.ApptDate != -1)
{
    //----- Retrieve value of DATE field.
    char szFldName[ILTR_MAX_FLDNAME];
    char szDate[20];
    long len = (long) sizeof(szDate);

    //----- read Date Field value.  If unreadable or null we can't filter.
    //----- don't complain, just leave with return code un-altered.
    ILFldName (tr, ILTR_map.ApptDate, szFldName, ILTR_MAX_FLDNAME);
    rc2 = ILTRGetField (tr, szFldName, szDate, &len);
    if (rc2 != SUCCESS || IL_STRLEN(szDate) == 0)
        return nRc;

    /*-----
    * Convert text or alpha date to number.
    * If the field is of type TEXT, convert it from
    * display format.  Otherwise, assume that it is
    * stored in IntelliLink "alpha" DATE format (e.g. 19961231).
    *-----*/
    if (ILTR_cDateType == ILX_TYPE_TEXT)
    {
        int month, day, year;
        rc2 = IL_DisplayToDate (szDate, &month, &day, &year);
        if (rc2 == SUCCESS)
            rc2 = IL_DateEncode (month, day, year, &lStartDate);

        if (rc2 != SUCCESS)
            //----- Invalid Date:  cannot do Date Range analysis or Filtering
            return nRc;
    }
    else
    {
        if (IL_AlphaDateOK (szDate))
            lStartDate = IL_AlphaToCodeDate (szDate);
        else
            //----- Invalid Date:  cannot do Date Range analysis or Filtering
            return nRc;
    }

    //----- Now compare start date with current date.
    if (ILTR_nSchOpt == ILTR_RANGE_FUTURE && lStartDate < ILTR_nDate)
        bSkipRecord = TRUE;

    //----- Verify that date is within specified range
    if (ILTR_nLoDate && ILTR_nHiDate)
        if (lStartDate < ILTR_nLoDate || lStartDate > ILTR_nHiDate)
            bSkipRecord = TRUE;
} //----- else if (ILTR_map.ApptDate != -1)

//----- The skip/no-skip decision has been made...
if (bSkipRecord)
{
    nRc |= ILTR_SKIP_WRITE;
    ILTR_action = ILTR_ACT_RANGE;
}

//----- Keep track of earliest and latest start date (repeating)
else if (bRepeatAppt)
{
    //----- Keep track of lowest and highest dates processed.
    if (ILTR_lEarliestDate == 0)
        ILTR_lEarliestDate = rpt.startDate;
    else if (rpt.startDate != 0 && rpt.startDate < ILTR_lEarliestDate)
        ILTR_lEarliestDate = rpt.startDate;
    if (rpt.startDate > ILTR_lLatestDate)
        ILTR_lLatestDate = rpt.startDate;
}

```

```

    if (rpt.stopDate > ILTR_lLatestDate)
        ILTR_lLatestDate = rpt.stopDate;

    //----- For unbounded repeats, give it a generous 10 year window
    if (rpt.stopDate == -1 &&
        ILTR_lLatestDate < ILTR_lEarliestDate + 10*365)
        ILTR_lLatestDate = ILTR_lEarliestDate + 10*365;
}

//----- Keep track of earliest and latest start date (non-repeating)
else
{
    //----- Is this the earliest start date?
    if (ILTR_lEarliestDate == 0)
        ILTR_lEarliestDate = lStartDate;

    else if (lStartDate != 0 && lStartDate < ILTR_lEarliestDate)
        ILTR_lEarliestDate = lStartDate;

    //----- Is this the latest start date?
    if (lStartDate > ILTR_lLatestDate)
        ILTR_lLatestDate = lStartDate;
}

return nRc;
} //---- ExportCheckApptDate

/*-----
 * Name:      ExportApplyUserFilter (called from ExportApplyFilters)
 *-----*/
static int ExportApplyUserFilter (ILTR_PTRANSL tr, int nRc)
{
    //---- Note that nRc is passed IN, then returned OUT

    BOOLEAN bFilterRecord = FALSE;          // Should record be filtered?

    //----- Filters not currently supported on the MAC
    #ifndef ILMAC
        //----- Is a filter currently active?
        if (ILTR_nFilterID != -1)
        {
            //----- Check if the record passes filter conditions.
            bFilterRecord = ILFilterRecord (tr, ILTR_nFilterID);
        }
    #endif

    if (bFilterRecord)
    {
        //----- Record has failed filter conditions.
        nRc |= ILTR_SKIP_WRITE;
        ILTR_action = ILTR_ACT_FILTER;
    }

    return nRc;
} //---- ExportApplyUserFilter

/*-----
 * Name:      ExportAddLogEntry (called from ExportWhile)
 *-----*/
static int ExportAddLogEntry (ILTR_PTRANSL tr)
{
    int i;                                // Loop variable
    int nResID;                            // Resource ID number
    unsigned long lAttrib;                 // Field attributes
    char cFldType;                         // Type of field
    char szFldName[ILTR_MAX_FLDNAME];      // Field name
    IL_PSTR lpMatch;                       // Pointer to character

    //----- Format action type in log record.
    IL_SYNC_MEM (ILTR_hRes.hNdx, ILTR_hRes.pNdx);
    switch (ILTR_action)
    {

```

```

case ILTR_ACT_ADD:
    nResID = ILTR_MSG_ADD;
    break;
case ILTR_ACT_DELETE:
    nResID = ILTR_MSG_DELETE;
    break;
case ILTR_ACT_FAN:
    nResID = ILTR_MSG_FAN;
    break;
case ILTR_ACT_FILTER:
    nResID = ILTR_MSG_FILTER;
    break;
case ILTR_ACT_IGNORE:
    nResID = ILTR_MSG_IGNORE;
    break;
case ILTR_ACT_RANGE:
    nResID = ILTR_MSG_RANGE;
    break;
case ILTR_ACT_READ:
    nResID = ILTR_MSG_READ;
    break;
case ILTR_ACT_REPLACE:
    nResID = ILTR_MSG_REPLACE;
    break;
case ILTR_ACT_SKIP:
    nResID = ILTR_MSG_SKIP;
    break;
case ILTR_ACT_UPDATE:
    nResID = ILTR_MSG_UPDATE;
}

//----- Write out log record.
if (ILTR_action != ILTR_ACT_SKIP)
{
    //----- If record title is typed, convert it
    ILFldName (tr, ILTR_map.ShowSource, szFldName, ILTR_MAX_FLDNAME);
    ILFldType (tr, szFldName, &cFldType, &lAttrib);

    if (cFldType == ILX_TYPE_DATE)
    {
        long lTempDate;

        lTempDate = IL_AlphaToCodeDate (ILTR_szRecName);
        IL_CodeDateToDisplay (lTempDate, ILTR_szRecName);
    }
    else if (cFldType == ILX_TYPE_TIME)
    {
        long lTempTime;

        lTempTime = IL_AlphaToCodeTime (ILTR_szRecName);
        IL_CodeTimeToDisplay (lTempTime, ILTR_szRecName);
    }

    //----- Set the record title to "<Unspecified>" if none supplied.
    if (IL_STRING_IS_NULL(ILTR_szRecName))
        ILSTMakeString (&ILTR_hRes, ILX_MSG_UNSPEC,
                        ILTR_szRecName, MAX_MSG);

    /*-----
    * Truncate the name at end of first line (only relevant to
    * multi-item fields).
    *-----*/
    lpMatch = IL_STRSTR (ILTR_szRecName, ILTR_szLineTerm);
    if (lpMatch != NULL)
        *lpMatch = '\0';

    //----- Truncate the name at EOS char if found
    lpMatch = IL_STRCHR (ILTR_szRecName, ILTR_EOS_CHAR);
    if (lpMatch != NULL)
        *lpMatch = '\0';

    //----- Place record and action in log file.
    if (ILAppendLog ( ILTR_hLog,
                      ILTR_hRes,
                      nResID,
                      ILTR_szRecName,

```

```

        NULL ))
    return ILTR_ERR_LOGFILE;

//----- Place field errors in log file.
for (i = 0; i < ILTR_nFldErrorNum; i++)
{
    //----- Convert internal to external field name.
    if (ILFldInToEx ( tr,
                     ILTR_fldError[i].szField,
                     szFldName,
                     ILTR_MAX_FLDNAME ))
        IL_STRCPY (szFldName, ILTR_fldError[i].szField);

    //----- Place next field error in log file.
    if (ILAppendLog ( ILTR_hLog,
                     ILTR_hRes,
                     ILTR_fldError[i].nError,
                     szFldName,
                     NULL) )
        return ILTR_ERR_LOGFILE;
}

//----- Exit without error.
return SUCCESS;

} //----- ExportAddLogEntry

/*-----
 * Name:      ExportEnd
 *-----*/
static int ExportEnd (ILTR_PTRANSL tr)
{
    int nRc;
    int rc2;
    char szMsgText[MAX_MSG];

    // secondary, ignored, return code
    // Display text

    //----- if Export is not initialized, simply return
    if (ILTR_bExpInitialized == FALSE)
        return SUCCESS;

    /*-----
    * For ILX_V3 only, set the lowest and highest dates.
    * This is an efficiency measure which minimizes the amount of
    * work that we need to do on the IMPORT side. We only need to
    * look for Target Records within the date range where all the
    * Source Records exist. Unfortunately ILX_V4 does ExportFromTarget
    * BEFORE it does ExportFromSource, so it loses this efficiency
    * feature. For one-way translations we could re-gain this efficiency
    * by changing the order of ILX_V4 operations. But for synchronization
    * we cannot do any such asymmetric range truncation.
    *-----*/
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        if (ILTR_nLoDate == 0 || ILTR_lEarliestDate > ILTR_nLoDate)
            ILTR_nLoDate = ILTR_lEarliestDate;

        if ( (ILTR_nHiDate == 0)
            || (ILTR_lLatestDate != 0 && ILTR_lLatestDate < ILTR_nHiDate) )
            ILTR_nHiDate = ILTR_lLatestDate;
    }

    //----- Show Please Wait message.
    ILSTMakeString (&ILTR_hRes, ILTR_MSG_WAIT, szMsgText, MAX_MSG);
    ILSetProgressText (tr, szMsgText);

    //----- Call callback function (if any) to terminate PIM function.
    if (ILTR_cbEnd == NULL)
        nRc = SUCCESS;
    else
        nRc = (*ILTR_cbEnd) (tr, &ILTR_appData);

    //----- Place last record in log file.
    if (ILTR_log)
    {

```

```

int resnum;
char szTemp[20];

if (ILTR_nSynchronize)
{
    if (ILTR_phase == ILTR_PHASE10)
        resnum = ILTR_MSG_TOTAL_INPUT_1ST; // count read from 1st system
    else
        resnum = ILTR_MSG_TOTAL_INPUT_2ND; // count read from 2nd system
    }
else
{
    if (ILTR_phase == ILTR_PHASE10)
        resnum = ILTR_MSG_TOTARG; // # of records exported from TARGET
    else
        resnum = ILTR_MSG_TOTSRC; // # of records exported from SOURCE
    }

IL_SPRINTF (szTemp, "%d", ILTR_recNum);
if (ILAppendLog (ILTR_hLog, ILTR_hRes, resnum, szTemp, NULL))
    if (nRc == SUCCESS)
        nRc = ILTR_ERR_LOGFILE;
}

if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
{
    //----- Close intermediate file.
    ILIFClose (ILTR_view);
    ILIFTerminate ();

    //----- Free memory used for ILIF "play area"
    if (ILTR_pILIF_Globals != NULL)
        IL_FREE_AND_ZERO (ILTR_hILIF_Globals, ILTR_pILIF_Globals);
}

else if (ILTR_pILTIF != NULL)
{
    //---- ILX_V4: put TIF back to sleep
    rc2 = ILTR_ILTIFCloseFileTemporarily (tr);
    if (nRc == SUCCESS)
        //---- report TIF error if it won't mask any other error
        nRc = rc2;
}

ILTR_bExpInitialized = FALSE;

#ifdef ILWIN
    //----- Remove environment ID window property
    RemoveProp (ILTR_hParentWin, ILTR_ENV_PROP);
#endif

//----- Dump field mapping to log file if appropriate
if (ILTR_rc &&
    ((ILTR_rc != ILTR_ERR_NOECS) ||
     (ILTR_nSynchronize == ILXTR_SYNC_NO)))
    ILDumpFieldsToLog (tr, ILTR_hLog);

//----- Free all field list, field mapping and character mapping tables.
ILFreeTables (tr);

//---- Free the field buffer
if (ILTR_field.handle != NULL)
    IL_FREE_AND_ZERO (ILTR_field.handle, ILTR_field.buffer);

//----- Filters not currently supported on the MAC
#ifdef ILMAC
    //----- Close filter file and free filter.
    if (ILTR_nFilterID != -1)
        ILFilterCleanup (tr, &ILTR_hFlt);

    //----- Was a filter active?
    if (ILTR_nFilterID != -1)
    {
        if (ILTR_FilterFieldBuffer.handle != NULL)
            IL_FREE_AND_ZERO (ILTR_FilterFieldBuffer.handle,
                             ILTR_FilterFieldBuffer.buffer);
    }

```

```

    }
#endif

//----- Signal end of import/export operation to progress window
ILStatusEnd (tr);

//----- pass back any error code
return nRc;

} //----- ExportEnd

/*-----
 * ExportRecord -- local function
 *
 * this function either calls ILIFPutRecord or ILTIFPutRecord
 *
 * For Phase 1 (export-before-import, from TARGET app),
 *     simply write record into ILTIF
 * For Phase 2 (export from SOURCE app),
 *     do field mapping before writing into ILTIF
 *-----*/
static int ExportRecord (ILTR_PTRANS� tr)
{
    int rc;

    /*-----
     * Now we put the record either to ILIF or to TIF.  The tr structure member
     * ILTR_phase tells us which to use.
     *-----*/
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);
        rc = ILIFPutRecord (ILTR_view, ILTR_rec.buffer, ILTR_rec.width);
    }
    else
        rc = ILTR_ILTIFPutRecord(tr);

    return rc;
} //----- ExportRecord

```



```

/*-----
* Name:      IMPORT.C
* Part of:   IntelliLink Translation Harness (ILTR)
* Purpose:   Contains top-level "driver" entrypoint, ILImport, for import
*            of record-oriented data.
*
* Functions:
*            ILImport
*            ImportBegin
*            ImportInit1
*            ImportInit2
*            ImportWhile
*            ImportCheckApptDate
*            ImportAddLogHeader
*            ImportAddLogEntry
*            ImportEnd
*            GetNextRecord
*            GetNextRecordFromILIF
*            GetNextRecordFromTIF
*            GetStartDate
*
* While Importing under ILWIN.EXE, ILWIN\xlatew.c\LoadTranslator sets
* ILTR_pDriver to point to ILImport. Then ILX_V3\dlgprog.c\ILX_DlgProgress
* calls (*ILTR_pDriver) repeatedly.
* (NOTE: Message loop is in ILWIN\xlatew.c\LoadTranslator.)
*
* While Importing under Windows ILX, ILX_V3\xlate.c\doTranslate calls
* loadxltr.c\ILX_LoadTranslator, which sets ILTR_pDriver to point to ILImport.
* Then ILX_V3\dlgprog.c\ILX_DlgProgress calls (*ILTR_pDriver) repeatedly.
* (NOTE: Message loop is in ILX_LoadTranslator.)
*
* While Importing on Macintosh, ILX_V3\xlate.c\doTranslate
* calls ILX_V3\macdrive.cpp\ILX_LoadTranslator, which
* calls ILTR\macstub.c\CallCodeResource for function=ILX_CALL_IMPORT, which
* calls ILTR\macstub.c\RunXlator with pXlator=&ILImport. Then RunXlator
* calls *pXlator repeatedly.
*
* NOTE:      this module is force-linked into every record-oriented translator
*
* Input:     Pointer to translator record
* Return:    SUCCESS, FAILURE, or error code
* Author:    Mike Blanchette, Copyright (c) IntelliLink, 1992-1995
*-----*/

#include "ilxapi.h"                // Includes ALL common headers

// for now WP stuff is only on the MAC
#ifdef ILMAC
#include "iltrwp.h"
#endif

// If using MFC with MSVC 4.1 we need to call AfxGetInstanceHandle
#ifdef ILMFC41
#include "afxwin.h"
#endif // ILMFC41

static int ImportBegin              (ILTR_PTRANSL tr);
static int ImportInit1              (ILTR_PTRANSL tr);
static int ImportInit2              (ILTR_PTRANSL tr);
static int ImportWhile              (ILTR_PTRANSL tr);
static int ImportCheckApptDate      (ILTR_PTRANSL tr, BOOLEAN *pbSkip);
static int ImportAddLogHeader        (ILTR_PTRANSL tr);
static int ImportAddLogEntry         (ILTR_PTRANSL tr);
static int ImportEnd                (ILTR_PTRANSL tr);
static int GetNextRecord             (ILTR_PTRANSL tr);
static int GetNextRecordFromILIF     (ILTR_PTRANSL tr);
static int GetNextRecordFromTIF      (ILTR_PTRANSL tr);
static int GetStartDate              (ILTR_PTRANSL tr, long IL_DIST *plDate);

/*-----
* Name:      ILImport
*-----*/
int IL_DECL EXP ILImport (ILTR_PTRANSL tr)
{
    int nRc;

```

```

// If no ddlMain was called we need an instance handle
#ifdef ILMFC41
    hXlatorInst = AfxGetInstanceHandle();
#endif // ILMFC41

/*-----
 * Word Processing sections use a different version of ILImport.
 * If this is a WP translation, then call ILImportWP.
 *-----*/
#ifdef ILMAC
    if (ILTR_nFunction == ILTB_SEC_MEMO)
    {
        nRc = ILImportWP (tr);
        return (nRc);
    }
#endif

switch (ILTR_nProcessStage)
{
    //----- Initialize translator.
    case ILTR_BEGIN:

        nRc = ImportBegin (tr);
        if (nRc == SUCCESS)
            nRc = ILProcessMessages (tr);
        if (nRc == SUCCESS)
            ILTR_nProcessStage = ILTR_WHILE;
        return nRc;

    //----- Import a record
    case ILTR_WHILE:

        /*-----
         * Phase30 is the conflict resolution and unload-to-target phase
         * of an ILX_V4 translation job. The 'WHILE' part of Phase30
         * is where the Target Translator gets a chance to "sanitize"
         * the source records before TIF does Conflict Analysis and
         * Resolution. But if the flag says skip it, we skip it.
         *-----*/
        if ( (ILTR_phase == ILTR_PHASE30)
            && (ILTR_Flags & ILTR_FLAGS_SKIP_SANITIZING_STEP) )
        {
            ILTR_nProcessStage = ILTR_END;
            return SUCCESS;
        }

        /*-----
         * Phase40 of Synchronization is where TIF records are unloaded
         * by the SOURCE translator. Unlike Phase30, where we let the
         * Target Translator sanitize source records, Phase40 does
         * no such thing, so we skip over the 'WHILE' phase and
         * jump straight to the END phase to unload the TIF file.
         *-----*/
        if (ILTR_phase == ILTR_PHASE40)
        {
            ILTR_nProcessStage = ILTR_END;
            return SUCCESS;
        }

        nRc = ImportWhile (tr);
        if (nRc == SUCCESS)
            nRc = ILProcessMessages (tr);
        else if (nRc == ILTR_EOF)
        {
            ILTR_nProcessStage = ILTR_END;
            nRc = ILProcessMessages (tr);
        }
        return nRc;

    //----- Shut down translator
    case ILTR_END:

        nRc = ImportEnd (tr);
        if (nRc == SUCCESS)
            nRc = ILProcessMessages (tr);
        return nRc;
}

```

```

        default:

            return ILERROR (ILTR_nProcessStage, ILTR_ERR_UNKNOWN);
    }

} //---- ILImport~

/*-----
* Name:      ImportBegin (called from ILImport)
*
* This function's behavior, in conjunction with the behavior of "dlgprog",
* implies the following basic cleanup rules for translator writers:
*
* (1) if your INIT returns an error, it must clean up any
*     allocated resources before doing so.
*
* (2) if your INIT returns SUCCESS, we guarantee that your END
*     routine will be called, so you can make your END routine
*     responsible for cleaning up any resources allocated by INIT.
*
* (3) But for historical reasons there is one more caveat:
*     If your BEGIN function returns ILTR_ERR_NODDE then before doing
*     so it must clean up any resources allocated by INIT.
*
*     When you return ILTR_ERR_NODDE, ILX_V3\dlgprog.c\ILX_DlgProgress
*     then tries to LAUNCH the target app, and if that succeeds then
*     your INIT and BEGIN functions will be called again.
*
* (4) Another peculiar case is ILTR_ERR_WAIT. But it is handled
*     differently. If your BEGIN return ILTR_ERR_WAIT then it will
*     be called again, but in this case your INIT will NOT be called
*     again.
*-----*/
static int ImportBegin (ILTR_PTRANSL tr)
{
    int nRc;

    //---- Do some initializations
    if (ILTR_bImpInitialized == FALSE)
    {
        nRc = ImportInit1 (tr);
        if (nRc != SUCCESS)
            return nRc;

        ILTR_bImpInitialized = TRUE;
        ILTR_bInitDone = FALSE;
        ILTR_bMustCleanUpBeforeQuitting = FALSE;
    }

    //---- Do app-specific initializations and some more initializations
    if (ILTR_bInitDone == FALSE)
    {
        //----- Call app-specific initialization function
        nRc = ILIniTranslator (tr, &ILTR_appData);
        if (nRc != SUCCESS)
            return nRc;                // Possibly ILTR_ERR_WAIT

        /*-----
        * Beyond this point we must always do cleanup, whether job
        * runs to completion, or fails prematurely, or is cancelled by
        * the user. Note that ILX\DLGPROG.C checks the 'MustCleanUp' flag.
        *-----*/
        ILTR_bMustCleanUpBeforeQuitting = TRUE;

        //---- do some more initializations
        nRc = ImportInit2 (tr);
        if (nRc != SUCCESS)
            return nRc;

        /*-----
        * Set flag to denote that INIT is done.
        * But beware of NODDE special case below!
        *-----*/
    }
}

```

```

    ILTR_bInitDone = TRUE;
}

if (ILTR_cbBegin != NULL)
{
    nRc = (*ILTR_cbBegin) (tr, &ILTR_appData);
    if (nRc != SUCCESS)
    {
        /*---- Special case for NODDE: force INIT to be called again!!
        if (nRc == ILTR_ERR_NODDE)
            ILTR_bInitDone = FALSE;

        return nRc;
    }
}

/*----- Set total record in progress bar control.
ILStatusImport (tr, ILTR_nRecords);

/*----- Write out record to log file using strings in resource file.
if (ILTR_log)
{
    nRc = ImportAddLogHeader (tr);
    if (nRc != SUCCESS)
        return nRc;
}

/*----- Is this the first of two passes on the source data?
if ( (ILTR_nSectionAttribs & ILTB_ATT_TWOPASS)
    && (ILTR_nPass == 1)
    && (ILTR_cbProgress == NULL) )
{
    char szText[MAX_MSG];                // Message text

    /*----- Display message for duration of first pass.
    ILSTMakeString ( &ILTR_hRes,
                    ILTR_MSG_FIRSTPASS,
                    szText,
                    MAX_MSG,
                    ILTR_szAppFile );

    /*----- Update the progress window.
    ILSetProgressText (tr, szText);
}

return SUCCESS;
} /*----- ImportBegin

/*-----
* Name:      ImportInit1 (called from ImportBegin)
*-----*/
static int ImportInit1 (ILTR_PTRANSL tr)
{
    int nRc;
    char szText[MAX_MSG];                // Display text

    /*----- set signature for Heap Logging (if enabled)
    ILUT_MemSig (ILTR_szTarTrans);

    /*----- Signal start of import/export operation to progress window
    ILStatusBegin (tr);

    /*----- Show Please Wait message.
    ILSTMakeString (&ILTR_hRes, ILTR_MSG_WAIT, szText, MAX_MSG);
    ILSetProgressText (tr, szText);

    /*----- Clear variables and set default line terminator string.
    ILTR_recNum = 0;
    ILTR_lImportRecs = 0;
    IL_STRCPY (ILTR_szLineTerm, ILTR_DEFAULT_TERM);
    ILTR_appData = NULL;

    /*----- Set ILTIME global default date and time format

```

```

IL_InitStdDateFormat ();
IL_InitStdTimeFormat ();

//----- set ILTR date and time format
IL_InitDateFormat ();
IL_InitTimeFormat ();

//----- Set initial pass number.
ILTR_nPass = 1;

#ifdef ILWIN
    //----- Make environment ID available as a window property
    SetProp (ILTR_hParentWin, ILTR_ENV_PROP,
        (IL_HANDLE) ILTR_eEnvironment);
#endif

if (ILTR_pTmpBuf == NULL)
{
    //----- initialize reusable buffer ILTR_pTmpBuf
    IL_ALLOC_MEM (sizeof (ILUT_BUFFER), ILTR_hTmpBuf, ILTR_pTmpBuf);
    if (ILTR_pTmpBuf == NULL)
        return ILTR_ERR_NOMEM;

    nRc = ILUT_InitBuffer (ILTR_pTmpBuf, 0, 512, ILTR_MAX_FIELDLLENGTH+1);
    if (nRc != SUCCESS)
        return ILTR_ERR_NOMEM;
}

//----- if doing an ILIF-based job, set up memory for ILIF to use...
if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
{
    //----- Allocate "play area" for ILIF to use
    IL_ALLOC_MEM (sizeof (ILIF_GLOBALS),
        ILTR_hILIF_Globals, ILTR_pILIF_Globals);
    if (ILTR_pILIF_Globals == NULL)
        return ILTR_ERR_NOMEM;

    //----- Zero play area
    IL_MEMSET (ILTR_pILIF_Globals, 0, sizeof (ILIF_GLOBALS));
}

return SUCCESS;
} //----- ImportInit1

/*-----
 * Name:      ImportInit2 (called from ImportBegin)
 *-----*/
static int ImportInit2 (ILTR_PTRANSL tr)
{
    int nRc;
    int nCount;                // Record count
    int nSections;            // Number of sections in ILIF file
    char szSection[MAX_SECNAME_SIZE]; // Section name

    /*-----
    * Ensure that the PUT callback routine has been registered.
    * Otherwise return with error indication.
    *-----*/
    if (ILTR_cbPut == NULL)
        return ILTR_ERR_NOPUTCB;

    //----- Initialize all field and character mapping structures.
    nRc = ILSetupTables (tr);
    if (nRc)
        return nRc;

    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        //----- Initialize Intermediate File (ILIF) subsystem.
        nRc = ILIFInit ( ILIF_ACC_SINGLE_FILE,
            ILTR_szWorkFile,
            IL_ATTR_READ,
            ILTR_NOT_USED,
            (int IL_DIST *) &nSections );
    }
}

```

```

    if (nRc != SUCCESS)
        //----- ERROR - could not initialize ILIF
        return ILERROR (nRc, ILTR_ERR_NOIF);

    //----- Open the relevant section of intermediate file.
    IL_SPRINTF (szSection, "%d", ILTR_nFunction);
    ILTR_view = ILIFOpen (szSection);
    if (ILTR_view == NULL_ILIF)
        return ILTR_ERR_NOIF;

    //----- Get total number of available records in intermediate file
    nCount = ILIFNumRecords (ILTR_view);
    ILSetRecCount (tr, nCount);

    //----- Report error if no records to import.
    if (nCount == 0)
        return ILTR_ERR_NORECS;

} //----- if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)

else
{
    //----- ILX_V4: wake up TIF
    nRc = ILTR_ILTIFReopenFile (tr);
    if (nRc != SUCCESS)
        return nRc;

    //----- If we haven't been told to SKIP the "sanitizing" step
    if ((ILTR_Flags & ILTR_FLAGS_SKIP_SANITIZING_STEP) == 0)
    {
        /*-----
        * Target Translator is now expected to sanitize the source
        * records. Find out how many source records there are.
        *-----*/
        INT32 lRecordCount = 0;;
        nRc = ILTR_ILTIFHowManyRecords (tr, &lRecordCount);
        if (nRc != SUCCESS)
            return nRc;

        nCount = (int) lRecordCount;
        ILSetRecCount (tr, nCount);

        /*-----
        * Report error if no records to import.
        * But don't complain when doing synchronization
        *-----*/
        if ((nCount == 0) && (ILTR_nSynchronize == ILXTR_SYNC_NO))
            return ILTR_ERR_NORECS;
    }
}

//----- Allocate memory for field buffer of default size.
ILTR_field.width = MAX_FIELD_LEN;
IL_ALLOC_MEM (ILTR_field.width, ILTR_field.handle, ILTR_field.buffer);
if (ILTR_field.buffer == NULL)
    return ILTR_ERR_NOMEM;

//----- Force new file to be created if zero length.
if (ILUT_IsZeroFile (ILTR_szAppFile))
    ILTR_nAttribs |= ILTB_ATT_REMFILE;

//----- Go create the file or database IFF it doesn't already exist.
if ( !ILTR_nFileExists ||
    (ILTR_nFileExists && (ILTR_nAttribs & ILTB_ATT_REMFILE)) )
    if (ILTR_cbNew != NULL)
        if ((nRc = (*ILTR_cbNew) (tr, &ILTR_appData))
            return nRc;

return SUCCESS;

} //----- ImportInit2

/*-----
* Name:      ImportWhile (called from ILImport)
*-----*/

```

```

static int ImportWhile (ILTR_PTRANSL tr)
{
    int nRc;
    BOOLEAN bFirstPass;           // Is this the first of two passes?
    BOOLEAN bSkipRecord = FALSE;  // Should record be skipped?
    char szFldName[ILTR_MAX_FLDNAME]; // Field name

    /*-----
    * Get next record, from ILIF or from TIF, and
    * let result of 'GetNextRecord' tell us what to do next.
    *-----*/
    nRc = GetNextRecord (tr);
    if (nRc != SUCCESS)
        return nRc; //---- ILTR_EOF or abnormal error

    //----- Synchronize pointers.
    IL_SYNC_MEM (ILTR_list.handle, ILTR_list.Name);
    IL_SYNC_MEM (ILTR_field.handle, ILTR_field.buffer);

    /*-----
    * If appointment falls outside desired range, skip it
    *-----*/
    if (ILTR_nFunction == ILTR_APPT && ILTR_action != ILTR_ACT_SKIP)
    {
        nRc = ImportCheckApptDate (tr, &bSkipRecord);
        if (nRc != SUCCESS)
            return nRc;
    }

    if (bSkipRecord)
        //----- Record has failed Date Range check.
        ILTR_action = ILTR_ACT_RANGE;
    else
    {
        //----- In Date Range - invoke callback function to import data record.
        ILTR_action = ILTR_ACT_ADD;
        nRc = (*ILTR_cbPut) (tr, &ILTR_appData);
        if (nRc > ILTR_SKIP_ALL || nRc < 0)
            return nRc;
    }

    /*-----
    * Has the record title been set? If not, attempt to
    * fetch default value from "show" field specified in map.
    *-----*/
    if (ILTR_szRecName[0] == ILTR_NULL_CHAR)
    {
        if (ILTR_map.ShowTarget != -1)
        {
            int rc2; // return code that we ignore

            ILFldName (tr, ILTR_map.ShowTarget, szFldName, ILTR_MAX_FLDNAME);
            if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
            {
                //--- do field mapping to get field value; ignore errors
                unsigned int nLen = MAX_MSG;
                rc2 = ILFldGet (tr, szFldName, ILTR_szRecName, &nLen);
            }
            else if (ILTR_phase == ILTR_PHASE30)
            {
                //--- fields are already mapped; just get value; ignore errors
                long len = (long) MAX_MSG;
                rc2 = ILTRGetField (tr, szFldName, ILTR_szRecName, &len);
            }
            else
                return ILERROR ((int) ILTR_phase, ILTR_ERR_INTERNAL);
        }
    }

    /*-----
    * Determine whether or not we're in the first pass of a multi-pass
    * translation (if so, logging is skipped in first pass).
    *-----*/
    bFirstPass =
        ((ILTR_nSectionAttribs & ILTB_ATT_TWOPASS) && ILTR_nPass == 1);

```

```

//----- Update progress bar.
if (!bFirstPass && !(nRc & ILTR_SKIP_METER))
    ILShowProgress (tr);

/*-----
 * Has translator asked to skip showing item?
 * If not, update the log file.
 *-----*/
if (!(nRc & ILTR_SKIP_LOG) && ILTR_log && !bFirstPass)
{
    int rc2 = ImportAddLogEntry (tr);
    if (rc2 != SUCCESS)
        return rc2;
}

return SUCCESS;
} //---- ImportWhile

/*-----
 * Name:      ImportCheckApptDate (called from ImportWhile)
 *-----*/
static int ImportCheckApptDate (ILTR_PTRANSI tr, BOOLEAN *pbSkip)
{
    BOOLEAN bSkipRecord = FALSE;          // Should record be skipped?
    ILTR_REPEAT rpt;
    long lStart;
    int rc;

    //----- Is this a repeating appointment? (check, and get repeat structure)
    if (ILIsRepeat2 (tr, &rpt))
    {
        //----- Ensure that dates are not past.
        if (ILTR_nSchOpt == ILTR_RANGE_FUTURE && rpt.startDate < ILTR_nDate)
            if (rpt.stopDate != -1 && rpt.stopDate < ILTR_nDate)
                bSkipRecord = TRUE;

        //----- Skip repeating items out of range
        if (ILTR_nLoDate && ILTR_nHiDate)
        {
            rc = ILItemHasInstancesInDateRange ( tr,          //---- see unfold.c
                                                  ILTR_nLoDate,
                                                  ILTR_nHiDate,
                                                  &rpt );

            //----- Free up exclusion list
            if (rpt.numExDates > 0)
            {
                IL_FREE_MEM_AND_ZERO_HANDLE (rpt.hExDates, rpt.exDates);
            }
            if (rc == FALSE)
                bSkipRecord = TRUE;
            else if (rc != TRUE)
                return rc;          // abnormal error
        }
    }

    //----- Processing an individual appointment.
    else if (ILTR_map.ApptDate != -1)
    {
        //----- Retrieve value of DATE field
        rc = GetStartDate (tr, &lStart);    // ignore rc

        //----- Now compare start date with current date.
        if (ILTR_nSchOpt == ILTR_RANGE_FUTURE && lStart < ILTR_nDate)
            bSkipRecord = TRUE;

        //----- Verify that date is within specified range
        if (ILTR_nLoDate && ILTR_nHiDate)
            if (lStart < ILTR_nLoDate || lStart > ILTR_nHiDate)
                bSkipRecord = TRUE;
    }

    *pbSkip = bSkipRecord;
    return SUCCESS;
}

```



```

} //---- ImportCheckApptDate

/*-----
 * Name:      ImportAddLogHeader (called from ImportBegin)
 *-----*/
static int ImportAddLogHeader (ILTR_PTRANSL tr)
{
    int nResID;

    IL_SYNC_MEM (ILTR_hRes.hNdx, ILTR_hRes.pNdx);

    /*-----
     * For Synchronization we always say that we're applying updates to
     * one of the systems -- either the 1st system or the 2nd system
     *-----*/
    if (ILTR_nSynchronize)
    {
        if (ILTR_phase == ILTR_PHASE30)
            // updating the so-called TARGET system -- don't add header yet!!
            return SUCCESS;
        else
            nResID = ILTR_MSG_UPDATE_2ND; // updating the so-called SOURCE system
    }

    /*-----
     * For SmartMerge we're either UPDATING or CREATING the TARGET file.
     * The test here is pretty crude. For non-file-based Desktop Apps we're
     * likely to say CREATE when UPDATE would be more appropriate. What we
     * really need to do is look at the ACCESS TYPE and maybe the
     * TOTAL_REBUILD system attribute. Maybe someday...
     *-----*/
    else if (ILTR_nFileExists && !(ILTR_nAttribs & ILTB_ATT_REMFILE))
        nResID = ILTR_MSG_UPDTAR;
    else
        nResID = ILTR_MSG_CRETAR;

    if (ILAppendLog (ILTR_hLog, ILTR_hRes, nResID, NULL, NULL))
        return ILTR_ERR_LOGFILE;

    return SUCCESS;
} //---- ImportAddLogHeader

/*-----
 * Name:      ImportAddLogEntry (called from ImportWhile)
 *-----*/
static int ImportAddLogEntry (ILTR_PTRANSL tr)
{
    int i; // Loop variable
    int nResID; // Resource ID number
    unsigned long lAttrib; // Field attributes
    char cFldType; // Type of field
    char szFldName[ILTR_MAX_FLDNAME]; // Field name
    IL_PSTR lpMatch; // Pointer to character

    //----- Format action type in log record.
    IL_SYNC_MEM (ILTR_hRes.hNdx, ILTR_hRes.pNdx);
    switch (ILTR_action)
    {
        case ILTR_ACT_ADD:
            ILTR_lImportRecs++;
            nResID = ILTR_MSG_ADD;
            break;
        case ILTR_ACT_DELETE:
            ILTR_lImportRecs++;
            nResID = ILTR_MSG_DELETE;
            break;
        case ILTR_ACT_FAN:
            ILTR_lImportRecs++;
            nResID = ILTR_MSG_FAN;
            break;
        case ILTR_ACT_FILTER:
            nResID = ILTR_MSG_FILTER;
            break;
    }
}

```

```

    case ILTR_ACT_IGNORE:
        nResID = ILTR_MSG_IGNORE;
        break;
    case ILTR_ACT_RANGE:
        nResID = ILTR_MSG_RANGE;
        break;
    case ILTR_ACT_READ:
        nResID = ILTR_MSG_READ;
        break;
    case ILTR_ACT_REPLACE:
        ILTR_lImportRecs++;
        nResID = ILTR_MSG_REPLACE;
        break;
    case ILTR_ACT_SKIP:
        nResID = ILTR_MSG_SKIP;
        break;
    case ILTR_ACT_UPDATE:
        ILTR_lImportRecs++;
        nResID = ILTR_MSG_UPDATE;
        break;

    case ILTR_ACT_LOADED_INTO_TIF:

        //---- don't log when loading TIF, except if there are errors
        if (ILTR_nFldErrorNum == 0)
            return SUCCESS;
        else
            nResID = ILTR_MSG_READ;
    }

    //----- If record title is typed, convert it
    ILFldName (tr, ILTR_map.ShowTarget, szFldName, ILTR_MAX_FLDNAME);
    ILFldType (tr, szFldName, &cFldType, &lAttrib);

    if (cFldType == ILX_TYPE_DATE)
    {
        long lTempDate;

        lTempDate = IL_AlphaToCodeDate (ILTR_szRecName);
        IL_CodeDateToDisplay (lTempDate, ILTR_szRecName);
    }
    else if (cFldType == ILX_TYPE_TIME)
    {
        long lTempTime;

        lTempTime = IL_AlphaToCodeTime (ILTR_szRecName);
        IL_CodeTimeToDisplay (lTempTime, ILTR_szRecName);
    }

    //----- Set the record title to "<Unspecified> if one was not supplied.
    if (ILTR_szRecName[0] == '\0')
        ILSTMakeString (&ILTR_hRes, ILX_MSG_UNSPEC, ILTR_szRecName, MAX_MSG);

    /*-----
    * Truncate the name at end of first line (only relevant to
    * multi-item fields).
    *-----*/
    lpMatch = IL_STRSTR (ILTR_szRecName, ILTR_szLineTerm);
    if (lpMatch != NULL)
        *lpMatch = '\0';

    //----- Truncate the name at EOS char if it exists.
    lpMatch = IL_STRCHR (ILTR_szRecName, ILTR_EOS_CHAR);
    if (lpMatch != NULL)
        *lpMatch = '\0';

    //----- Write out log record.
    if (ILAppendLog (ILTR_hLog, ILTR_hRes, nResID, ILTR_szRecName, NULL))
        return ILTR_ERR_LOGFILE;

    //----- Place field errors in log file.
    for (i = 0; i < ILTR_nFldErrorNum; i++)
    {
        //----- Convert internal to external field name.
        if (ILFldInToEx (tr,
            ILTR_fldError[i].szField,

```

```

        szFldName,
        ILTR_MAX_FLDNAME ))
    IL_STRCPY (szFldName, ILTR_fldError[i].szField);

    //----- Place next field error in log file.
    if (ILAppendLog ( ILTR_hLog,
        ILTR_hRes,
        ILTR_fldError[i].nError,
        szFldName,
        NULL) )
        return ILTR_ERR_LOGFILE;
    }

    return SUCCESS;
} //----- ImportAddLogEntry

/*-----
 * Name:      ImportEnd
 *-----*/
static int ImportEnd (ILTR_PTRANSL tr)
{
    int nRc;
    int rc2;                      // secondary return code
    char szText[MAX_MSG];         // Message text

    //----- if Export is not initialized, simply return
    if (ILTR_bImpInitialized == FALSE)
        return SUCCESS;

    //----- Show Please Wait message.
    ILSTMakeString (&ILTR_hRes, ILTR_MSG_WAIT, szText, MAX_MSG);
    ILSetProgressText (tr, szText);

    //----- Call callback function (if any) to terminate PIM function.
    if (ILTR_cbEnd == NULL)
        nRc = SUCCESS;
    else
    {
        if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
            //----- For ILX_V4 TIF-based translation, we want ILTR_recNum to
            //----- count records unloaded, not records sanitized.
            ILTR_recNum = 0;

        if (ILTR_pILTIF != NULL)
            //----- For any TIF-based import, make sure count of imported
            //----- records doesn't include any sanitizing count (e.g. FANNING)
            ILTR_lImportRecs = 0;

        nRc = (*ILTR_cbEnd) (tr, &ILTR_appData);
    }

#ifdef ILWIN
    //----- Remove environment ID window property; it's no longer needed!
    RemoveProp (ILTR_hParentWin, ILTR_ENV_PROP);
#endif

    //----- Place last record in log file.
    if (ILTR_log)
    {
        char szTemp[20];           // Temporary string
        int resnum;

        if (ILTR_nSynchronize)
        {
            if (ILTR_phase == ILTR_PHASE30)
                resnum = ILTR_MSG_TOTAL_UPDATE_1ST; // updates applied to 1st system
            else
                resnum = ILTR_MSG_TOTAL_UPDATE_2ND; // updates applied to 2nd system
        }
        else
            resnum = ILTR_MSG_TOTARG; // # of records imported into TARGET

        IL_SPRINTF (szTemp, "%d", ILTR_recNum);
        if (ILAppendLog (ILTR_hLog, ILTR_hRes, resnum, szTemp, NULL))

```

```

        if (nRc == SUCCESS)
            nRc = ILTR_ERR_LOGFILE;
    }

    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        //----- Close intermediate file.
        ILIFClose (ILTR_view);
        ILIFTerminate ();

        //----- Free memory used for ILIF "play area"
        if (ILTR_pILIF_Globals != NULL)
            IL_FREE_AND_ZERO (ILTR_hILIF_Globals, ILTR_pILIF_Globals);
    }

    else if (ILTR_pILTIF != NULL)
    {
        //----- ILX_V4: put TIF back to sleep
        rc2 = ILTR_ILTIFCloseFileTemporarily (tr);
        if (nRc == SUCCESS)
            //----- report TIF error if it won't mask any other error
            nRc = rc2;
    }

    ILTR_bImpInitialized = FALSE;

    //----- Dump field mapping to log file when appropriate
    if (ILTR_nSynchronize == ILXTR_SYNC_NO ||
        ILTR_phase == ILTR_PHASE40 ||
        (ILTR_rc && (ILTR_rc != ILTR_ERR_NORECS)))
        ILDumpFieldsToLog (tr, ILTR_hLog);

    //----- Free all field list, field mapping and character mapping tables.
    ILFreeTables (tr);

    //----- Free the field buffer
    if (ILTR_field.handle != NULL)
        IL_FREE_AND_ZERO (ILTR_field.handle, ILTR_field.buffer);
    ILTR_field.width = 0;

    //----- Signal end of import/export operation to progress window
    ILStatusEnd (tr);

    //----- pass back any error code
    return nRc;
} //----- ImportEnd

/*-----
 * GetNextRecord
 *-----*/
static int GetNextRecord (ILTR_PTRANSL tr)
{
    int nRc;

    //----- Set default values for record name and action type.
    ILTR_szRecName[0] = '\0';
    ILTR_nFldErrorNum = 0;
    IL_MEMSET (ILTR_fldError, '\0', sizeof (ILTR_FLDERR) * ILTR_MAX_FLDERR);

    /*-----
    * Get a record -- from ILIF or from TIF
    * Also, for 2-pass translators, shift from 1st pass to 2nd
    * pass when 1st pass hits EOF.
    * Increment ILTR_nRecNum.
    *-----*/
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
        nRc = GetNextRecordFromILIF (tr);
    else
        nRc = GetNextRecordFromTIF (tr);

    if (nRc != ILTR_EOF)
        ILTR_recNum++;

    return nRc;
}

```

```

} //---- GetNextRecord

/*-----
 * GetNextRecordFromILIF
 * Possible return codes are:
 *   SUCCESS
 *   ILTR_EOF
 *   ILTR_ERR_NOMEM
 *   ILTR_ERR_RECORD_TOO_BIG
 *   ILTR_ERR_NOIF
 *-----*/
static int GetNextRecordFromILIF (ILTR_PTRANSI tr)
{
    int nRc;

NextRecord:
//-----

    IL_SYNC_MEM (ILTR_rec.handle, ILTR_rec.buffer);
    nRc = ILIFGetRecord (ILTR_view, ILTR_rec.buffer, ILTR_rec.width);

    if (nRc == ILIF_ERR_EOF)
    {
        //----- Found end of file.
        //----- Prepare for second pass if this is a two-pass translator.
        if ((ILTR_nSectionAttribs & ILTB_ATT_TWOPASS) && ILTR_nPass == 1)
        {
            char szSection[MAX_SECNAME_SIZE];    // Section name
            int nSections;

            //----- Increment the pass number.
            ILTR_recNum = 0;
            ILTR_nPass++;

            //----- Close the section and intermediate file.
            ILIFClose (ILTR_view);
            ILIFTerminate ();

            /*-----
             * Now re-open the intermediate file and section to
             * reposition at beginning.
             *-----*/
            nRc = ILIFInit ( ILIF_ACC_SINGLE_FILE,
                            ILTR_szWorkFile,
                            IL_ATTR_READ,
                            ILTR_NOT_USED,
                            (int IL_DIST *) &nSections );
            if (nRc != SUCCESS)
                return ILERROR (nRc, ILTR_ERR_NOIF);

            IL_SPRINTF (szSection, "%d", ILTR_nFunction);
            ILTR_view = ILIFOpen (szSection);
            if (ILTR_view == NULL_ILIF)
                return ILTR_ERR_NOIF;

            //----- Reset the bar for second import operation.
            ILStatusImport (tr, ILTR_nRecords);

            //----- Go back and read first intermediate record.
            goto NextRecord;
        }
        else
            return ILTR_EOF;
    }

    /*-----
     * Insufficient room in buffer to hold incoming record.
     * Reallocate the buffer size and retry.
     *-----*/
    while (nRc == ILIF_ERR_NOROOM)
    {
        /*-----
         * Increment buffer size and verify that it does not exceed

```

```

    * fixed limit.
    *-----*/
    ILTR_rec.width += MAX_IF_INCR;
    if (ILTR_rec.width > MAX_IF_ALLOC)
    {
        ILTR_rec.width -= MAX_IF_INCR; // back to currently allocated size
        return ILTR_ERR_RECORD_TOO_BIG;
    }

    //----- Reallocate the buffer to larger size.
    IL_REALLOC_MEM (ILTR_rec.width, ILTR_rec.handle, ILTR_rec.buffer);

    //----- Unable to allocate memory for larger buffer size.
    if (ILTR_rec.buffer == NULL)
        return ILTR_ERR_NOMEM;

    //----- Now try reading the record again in the new buffer.
    nRc = ILIFGetRecord (ILTR_view, ILTR_rec.buffer, ILTR_rec.width);
}

if (nRc != SUCCESS)
    //--- ERROR reading intermediate file record.
    return ILERROR (nRc, ILTR_ERR_NOIF);

return SUCCESS;
} //---- GetNextRecordFromILIF

/*-----
 * GetNextRecordFromTIF
 *-----*/
static int GetNextRecordFromTIF (ILTR_PTRANSI tr)
{
    int nRc = ILTR_ILTIFReadNextRecord (tr);
    if (nRc == TIF_EOF)
        return ILTR_EOF;
    else
        return nRc;
} //---- GetNextRecordFromTIF

/*-----
 * GetStartDate
 *-----*/
static int GetStartDate (ILTR_PTRANSI tr, long IL_DIST *pDate)
{
    int rc;
    int DateFieldIndex;
    ILTR_FLDPTR FieldList;
    IL_PSTR pFieldName;
    char cDateType;
    char szTmp[MAX_MSG];
    long len;

    //----- Get field list and field index needed for DATE field lookup
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        //---- when using ILIF, we use SOURCE fieldname to get date
        DateFieldIndex = ILTR_map.ApptDate;
        FieldList = ILTR_map.pSource;
    }
    else
    {
        //---- for ILX_V4 (using TIF as intermediate file)
        //---- we use TARGET fieldname to get date
        int SourceFieldIndex = ILTR_map.ApptDate;
        DateFieldIndex = ILTR_map.pSource[SourceFieldIndex].MapField;
        FieldList = ILTR_map.pTarget;
    }

    //---- Get field name and field type for DATE field
    pFieldName = FieldList[DateFieldIndex].IntName;
    cDateType = FieldList[DateFieldIndex].Type;

```

```
//---- Get field value from intermediate file
len = (long) sizeof(szTmp);
rc = ILTRGetField (tr, pFieldName, szTmp, &len); //rc is ignored!!

/*-----
 * Convert text or alpha date to number.
 * If the field is of type TEXT, convert it from
 * display format. Otherwise, assume that it is
 * stored in IntelliLink "alpha" DATE format (e.g. 19961231).
 *-----*/
if (cDateType == ILX_TYPE_TEXT)
    IL_DisplayToCodeDate (szTmp, plDate);
else
    *plDate = IL_AlphaToCodeDate (szTmp);

return SUCCESS;
} //---- GetStartDate
```

```

/*-----
* File:      ILXTRANS.CPP
*
* Purpose:   This is SAMPLE CODE, which demonstrates how to code
*            ILTR callback routines for an ILXTRANS.LIB-based translator.
*
* Functions: ILIniTranslator
*            ILXTransBegin
*            ILXTransGet
*            ILXTransPut
*            ILXTransPutNext
*            ILXTransEnd
*            ILXTransNew
* Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/

#include "ilxtrans.h"
#include "ilxterr.h"          // error codes & error handling protos
#include "cilglobl.h"         // header for CILGlobalData class
#include "cilfa.h"            // header for CILField class
#include "cilrec.h"           // header for CILRecord class
#include "cildata.h"          // header for CILDataStore class
#include "ciltrans.h"         // header for CILTranslator class

#include "dsample.h"          // sample derived CILDataStore class header

/*-----
* Function:  ILIniTranslator
* Purpose:   Initialization routine for all translators.  Its purpose is
*            to allocate a new translator object and register all callback
*            routines with the engine.  This routine will also initialize
*            portions of the translator object's data.
* Input:     tr --- pointer to ILTR translation structure
*            ppTrans -- pointer to pointer to CILTranslator object
* Return:    SUCCESS, ILTR_ERR_INVFUN, or ILTR_ERR_NOMEM
* Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
* Notes:     A GlobalData object is allocated here and a pointer to it is
*            stored in the m_pGlobalData member. A DataStore object is also
*            allocated and its pointer stored in the GlobalData object.
*-----*/
int IL_DECL ILIniTranslator
(
    ILTR_PTRANSL tr,
    IL_PANY IL_DIST * ppTrans)
{
    int iRc;          // return code
    CILTransPtr pTrans; // pointer to translator object

    //----- construct new translator object
    pTrans = (CILTranslator *) new CILTranslator;
    if (pTrans == NULL)
        return ILTR_ERR_NOMEM;

    //----- create the GlobalData object
    pTrans->m_pGlobalData = (CILGlobalData *) new CILGlobalData();

    //----- was it allocated?
    if (pTrans->m_pGlobalData == NULL)
        return ILTR_ERR_NOMEM;

    //----- call Initialization routine
    iRc = pTrans->Initialize (tr);
    if (iRc)
        return iRc;

    //----- register call back routines to the engine
    ILRegisterBeginCB (tr, (ILTR_CBFUN) ILXTransBegin);
    ILRegisterEndCB   (tr, (ILTR_CBFUN) ILXTransEnd);
    ILRegisterGetCB   (tr, (ILTR_CBFUN) ILXTransGet);
    ILRegisterPutCB   (tr, (ILTR_CBFUN) ILXTransPut);
    ILRegisterRepeatCB (tr, (ILTR_CBFUN) ILXTransPutNext);
    ILRegisterNewCB    (tr, (ILTR_CBFUN) ILXTransNew);

    //----- Save ptr to CILTrans ptr in ILTR translation struct by setting ppTrans
    *ppTrans = (void *) pTrans;
}

```



```

    //----- Initialization complete
    return SUCCESS;
}

/*-----
 * Function:  ILXTransBegin
 * Purpose:   Glue code between ILTR and the actual translator.
 *            This function has been registered as the begin callback in the
 *            translation engine. Its purpose is to call the Begin member
 *            function of the CILTranslator object which is passed via ppTrans.
 * Input:     tr --- pointer to ILTR translation structure
 *            ppTrans -- pointer to pointer to CILTranslator object
 * Return:    SUCCESS, ILTR_ERR_INVFUN, ILTR_ERR_CORRUPT_DATA, ILTR_ERR_FILE,
 *            or ILTR_ERR_NORECS
 * Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
 * Notes:     The FieldArray and Record objects are allocated here and pointers
 *            to them are stored in the GlobalData object.
 *-----*/
int IL_DECL ILXTransBegin
    (ILTR_PTRANSL tr,
     IL_PANY IL_DIST * ppTrans)
{
    int          iRc;                // Function return code
    CILTransPtr  pTrans;            // pointer to CILTranslator object
    CILGlPtr     gd;                // pointer to GlobalData object

    //----- Establish pointer to CILTranslator object
    pTrans = (CILTransPtr) *ppTrans;
    if (pTrans == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- Establish reference to GlobalData object
    gd = pTrans->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- Create the DataStore object
    pTrans->m_pGlobalData->m_pDataStore = (CILSampleData *) new CILSampleData();

    //----- was the DataStore created?
    if (pTrans->m_pGlobalData->m_pDataStore == NULL)
        return ILTR_ERR_NOMEM;

    //----- Create the FieldArray object
    gd->m_pFldArray = (CILFieldArray *) new CILFieldArray ();

    //----- Was FieldArray object created?
    if (gd->m_pFldArray == NULL)
        return ILTR_ERR_NOMEM;

    //----- create the record object, pass along field array
    gd->m_pRecord = (CILRecord *) new CILRecord (&gd->m_pFldArray);

    //----- was Record object created?
    if (gd->m_pRecord == NULL)
        return ILTR_ERR_NOMEM;

    //----- Initialize the DataStore object
    iRc = gd->m_pDataStore->Initialize (tr, gd);
    if (iRc)
        return ILERROR (iRc, iRc);

    //----- call the "Real" Begin routine
    iRc = pTrans->Begin (tr);

    //----- return value
    return iRc;
}

/*-----
 * Function:  ILXTransGet
 * Purpose:   Glue code between ILTR and the actual translator.
 *            This function has been registered as the Get callback in the

```

```

*      translation engine.  Its purpose is to call the Get member
*      function of the CILTranslator object which is passed via ppTrans.
*  Input:  tr --- pointer to ILTR translation structure
*          ppTrans -- pointer to pointer to CILTranslator object
*  Return:  SUCCESS, ILTR_ERR_INVFUN, ILTR_ERR_CORRUPT_DATA, ILTR_ERR_FILE,
*          or ILTR_ERR_NORECS
*  Author:  Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int IL_DECL ILXTransGet
    (ILTR_PTRANSL tr,
     IL_PANY IL_DIST * ppTrans)
{
    //----- local vars
    int      iRc;                // Function return code
    CILTransPtr pTrans;         // pointer to CILTranslator object

    //----- Establish pointer to CILTranslator object
    pTrans = (CILTransPtr) *ppTrans;
    if (pTrans == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- call the "Real" Get routine
    iRc = pTrans->Get (tr);

    //----- return value
    return iRc;
}

/*-----*/
*  Function:  ILXTransPut
*  Purpose:   Glue code between ILTR and the actual translator.
*             This function has been registered as the Put callback in the
*             translation engine.  Its purpose is to call the Put member
*             function of the CILTranslator object which is passed via ppTrans.
*  Input:    tr --- pointer to ILTR translation structure
*            ppTrans -- pointer to pointer to CILTranslator object
*  Return:    SUCCESS, ILTR_ERR_INVFUN, ILTR_ERR_CORRUPT_DATA, ILTR_ERR_FILE,
*            or ILTR_ERR_NORECS
*  Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int IL_DECL ILXTransPut
    (ILTR_PTRANSL tr,
     IL_PANY IL_DIST * ppTrans)
{
    //----- local vars
    int      iRc;                // Function return code
    CILTransPtr pTrans;         // pointer to CILTranslator object

    //----- Establish pointer to CILTranslator object
    pTrans = (CILTransPtr) *ppTrans;
    if (pTrans == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- call the "Real" Put routine
    iRc = pTrans->Put (tr);

    //----- return value
    return iRc;
}

/*-----*/
*  Function:  ILXTransPutNext
*  Purpose:   Glue code between ILTR and the actual translator.
*             This function has been registered as the Repeat callback in the
*             translation engine.  Its purpose is to call the PutNext member
*             function of the CILTranslator object which is passed via ppTrans.
*  Input:    tr --- pointer to ILTR translation structure
*            ppTrans -- pointer to pointer to CILTranslator object
*            pRepeat - pointer to repeat struct
*  Return:    SUCCESS, ILTR_ERR_INVFUN, ILTR_ERR_CORRUPT_DATA, ILTR_ERR_FILE,
*            or ILTR_ERR_NORECS
*  Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995

```

```

/*-----*/
int IL_DECL ILXTransPutNext
    (ILTR_PTRANSL tr,
     IL_PANY IL_DIST * ppTrans,
     ILTR_PREPEAT pRepeat)
{
    //----- local vars
    int iRc; // Function return code
    CILTransPtr pTrans; // pointer to CILTranslator object

    //----- Establish pointer to CILTranslator object
    pTrans = (CILTransPtr) *ppTrans;
    if (pTrans == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- call the "Real" PutNext routine
    iRc = pTrans->PutNext (tr, pRepeat);

    //----- return value
    return iRc;
}

/*-----*/
* Function: ILXTransEnd
* Purpose:  Glue code between ILTR and the actual translator.
*           This function has been registered as the End callback in the
*           translation engine. Its purpose is to call the End member
*           function of the CILTranslator object which is passed via ppTrans.
* Input:    tr --- pointer to ILTR translation structure
*           ppTrans -- pointer to pointer to CILTranslator object
* Return:    SUCCESS, ILTR_ERR_INVFUN, ILTR_ERR_CORRUPT_DATA, ILTR_ERR_FILE,
*           or ILTR_ERR_NORECS
* Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
/*-----*/
int IL_DECL ILXTransEnd
    (ILTR_PTRANSL tr,
     IL_PANY IL_DIST * ppTrans)
{
    //----- local vars
    int iRc; // Function return code
    CILTransPtr pTrans; // pointer to CILTranslator object
    CILGlPtr gd; // pointer to GlobalData object

    //----- Establish pointer to CILTranslator object
    pTrans = (CILTransPtr) *ppTrans;
    if (pTrans == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- reference global data
    gd = pTrans->m_pGlobalData;

    //----- call the "Real" End routine
    iRc = pTrans->End (tr);

    //----- destroy the record object
    delete (CILRecord *) gd->m_pRecord;

    //----- destroy the datastore
    delete (CILSampleData *) gd->m_pDataStore;

    //----- destroy the FieldArray
    delete (CILFieldArray *) gd->m_pFldArray;

    //----- free the GlobalData object
    delete gd;

    //----- translation is done: destroy translator object
    delete pTrans;

    //----- return value
    return iRc;
}

```

```

/*-----
* Function:  ILXTransNew
* Purpose:   Glue code between ILTR and the actual translator.
*           This function has been registered as the New callback in the
*           translation engine. Its purpose is to call the New member
*           function of the CILTranslator object which is passed via ppTrans.
* Input:     tr--- pointer to ILTR translation structure
*           ppTrans -- pointer to pointer to CILTranslator object
* Return:     SUCCESS, ILTR_ERR_INVFUN, ILTR_ERR_CORRUPT_DATA, ILTR_ERR_FILE,
*           or ILTR_ERR_NORECS
* Author:     Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int IL_DECL ILXTransNew
    (ILTR_PTRANSL tr,
     IL_PANY IL_DIST * ppTrans)
{
    //----- local vars
    int      iRc;                // Function return code
    CILTransPtr pTrans;          // pointer to CILTranslator object

    //----- Establish pointer to CILTranslator object
    pTrans = (CILTransPtr) *ppTrans;
    if (pTrans == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- call the "Real" New routine
    iRc = pTrans->New (tr);

    //----- return value
    return iRc;
}

```

```

#ifdef __CILTRANS
/*-----
 * Module: ciltrans.h
 * Purpose: Header file for generic Translator class
 * Author: Dave McConville, Copyright (c) IntelliLink, 1995
 * Notes: All objects are implemented with pure C++ in order to allow these
 *        class to be platform independent. No use of MFC or MacApp has been
 *        made.
 *-----*/
#define __CILTRANS // Indicate header inclusion

//----- Include all the relevant headers
#include "cilglobl.h"
#include "il提高.h"

/*-----
 * ILTranslator class.
 *-----*/
#define CILTransPtr CILTranslator *
class CILTranslator
{
public:
    //----- Class constructor
    CILTranslator ();

    //----- Class destructor
    ~CILTranslator ();

    /*-----
     * Caller accessible functions
     *-----*/

    virtual int Initialize // Initialization routine
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int Begin // Translator Begin processing
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int Get // Translator Get processing
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int PutRepeatToIL // wrapper for ILPutRepeat function
        ( ILTR_PTRANSL tr,
          ILTR_PREPEAT pRepeat,
          IL_PSTR szStartField,
          IL_PSTR szEndField );

    virtual int Put // Translator Put processing
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int PutNext // Translator Put a single record
        (ILTR_PTRANSL tr, // pointer to ILTR translator structure
         ILTR_PREPEAT pRepeat); // pointer to repeat info for this rec

    virtual int End // End call back function
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int New // New call back function
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int CreateTIF // Create TIF file and define TIF fields
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int LoadTIF // Load TIF file with existing data
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int UnloadTIF // UnLoad TIF records to data store
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int UnloadRecord // UnLoad a single TIF record
        (ILTR_PTRANSL tr); // pointer to ILTR translator structure

    virtual int UnloadInstance // UnLoad one fanned instance of an item
        (ILTR_PTRANSL tr, // pointer to ILTR translator structure

```

```
        ILTR_PREPEAT pRepeat);    // pointer to repeat info for this rec

virtual int OpenDataStore        // Open data store, handling passwords
(ILTR_PTRANSL tr,                // Pointer to ILTR translator structure
 CILGIPtr    gd,                // Pointer to GlobalData object
 int         nMode);            // Open mode (read or write)

virtual int FanBeforeUnload      // Fan all items that need to be fanned
(ILTR_PTRANSL tr);              // pointer to ILTR translator structure

//----- member attributes
CILGIPtr    m_pGlobalData; // pointer to global data structure
BOOLEAN     m_bNewFile;    // TRUE if processing new file

};

#endif // __CILTRANS
```

```

/*-----
* File:      CILTRANS.CPP
* Purpose:   This module contains the implementation for the member functions
*           of the CILTranslator class.
*
* Functions: CILTranslator::CILTranslator (constructor)
*           CILTranslator::~CILTranslator (destructor)
*           CILTranslator::Initialize
*           CILTranslator::Begin
*           CILTranslator::OpenDataStore
*           CILTranslator::Get
*           CILTranslator::Put
*           CILTranslator::PutNext
*           CILTranslator::End
*           CILTranslator::New
*           CILTranslator::CreateTIF
*           CILTranslator::LoadTIF
*           CILTranslator::UnloadTIF
*           CILTranslator::UnloadRecord
*           CILTranslator::UnloadInstance
*           CILTranslator::FanBeforeUnload
*           IL_OutputDebugLog
*           MakeLogMessage
*
* Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995-6
*-----*/

#include "cilglobl.h"      // header for CILGlobalData class
#include "ciltrans.h"      // header for CILTranslator class
#include "cilfa.h"         // header for CILField class
#include "cildata.h"       // header for CILDataStore class
#include "cilrec.h"        // header for CILRecord class
#include "ilxtrans.h"
#include "ilxterr.h"       // error codes & error handling protos

//----- Internal function for invoking the Choose Records dialog
static int CallChooseRecords (ILTR_PTRANSL tr);

//----- Type for ChooseRecords function pointer
typedef DLLEXPORT int (IL_DECL EXP *PCHOOSE) (ILTR_PTRANSL tr);

/*-----
* Function   : CILTranslator::CILTranslator
* Purpose    : Class constructor, initialize data members
* Parameters : none
* Returns    : none
*-----*/
CILTranslator::CILTranslator ()
{
    m_pGlobalData = NULL;    // global data not yet allocated
    m_bNewFile = FALSE;     // assume using existing file
};

/*-----
* Function   : CILTranslator::~CILTranslator
* Purpose    : Class destructor
* Parameters : none
* Returns    : none
*-----*/
CILTranslator::~CILTranslator ()
{
}

/*-----
* Function   : CILTranslator::Initialize
* Purpose    : The routine is provided to do initialization for the
*           translator.
* Parameters : tr --- pointer to ILTR translation structure
* Returns    : ILXT_OK - all is well,
*-----*/
int CILTranslator::Initialize (ILTR_PTRANSL tr)
{
    return ILXT_OK;
}

```

```

/*-----
 * Function:  CILTranslator::Begin
 * Purpose:   This routine is the implementation for the Begin member function
 *            of the CILTranslator class.  It is to be used as the Begin
 *            callback routine from ILTR.
 * Input:     tr --- pointer to ILTR translation structure
 * Return:    SUCCESS - all is well
 *            ILXT_ERR_USAGE - null pointer to global data passed
 *            ILTR_ERR_NOMEM - unable to allocate memory
 *            ILTR_ERR_NORECS - no records to export
 *            ILXT_ERR_TIF or ILXT_ERR_FAIL - abnormal TIF problems
 * Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
 *-----*/
int CILTranslator::Begin (ILTR_PTRANSL tr)
{
    CILFldArrayPtr pFieldArray;          // Pointer to field array object
    int             nFieldCount = 0;      // Number of fields in field array
    int             iRc;                  // Return code variable
    CILGlPtr        gd;                  // Pointer to GlobalData object
    CILDataStore    *pDataStore;          // Pointer to DataStore object
    BOOLEAN         bOpened;              // Was DataStore opened?

    //----- Establish pointer to application global
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- Fetch the DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- Fetch the FieldArray object from gd
    pFieldArray = gd->m_pFldArray;

    //----- Initialize FieldArray, allocate and init Field objects
    iRc = pFieldArray->Initialize (tr, gd);
    if (iRc)
        LOG_ERROR_AND_EXIT (iRc, iRc);

    /*-----
     * Determine if the translator needs to perform ToDo range checking.
     * Check must be performed on each record if this is a to do section
     * and a date range has been specified.  But we turn off date range
     * checking for SYNC and when doing export before import.
     *-----*/
    if ( ((ILTR_nFunction == ILTR_TODO) || (ILTR_nFunction == ILTR_CALL))
        && (ILTR_nLoDate != 0)
        && (ILTR_nHiDate != 0)
        && (ILTR_direction == ILTR_EXPORT)
        && (ILTR_nSynchronize == ILXTR_SYNC_NO)
        && (ILTR_phase != 10) )
        gd->m_bToDoRangeCheck = TRUE;

    /*-----
     * Determine if the translator should check for and skip done to do items.
     * We never skip done to do items when doing synchronization, and we
     * don't skip done todo items when doing "Export before Import".
     *-----*/
    if ( ((ILTR_nFunction == ILTR_TODO) || (ILTR_nFunction == ILTR_CALL))
        && (ILTR_direction == ILTR_EXPORT)
        && (ILTR_nSynchronize == ILXTR_SYNC_NO)
        && (ILTR_phase != 10) )
        gd->m_bToDoDoneCheck = TRUE;

    /*-----
     * Allocate the record and field buffers storing pointers and handles
     * in GlobalData.  The size of these buffers is determined by the data
     * store.  It is okay for the datastore to specify a zero length record
     * buffer in the event a record buffer is not needed.
     *-----*/

    //----- Do we need to allocate a record buffer?
    if (pDataStore->m_uiInitRecBufSize)
    {

```



```

//----- Allocate it
gd->m_lpRecBuf = (IL_PSTR) IL_ALLOC (pDataStore->m_uiInitRecBufSize,
                                     gd->m_hRecBuf);

//----- Was record buffer allocated?
if (gd->m_lpRecBuf == NULL)
    EXIT_WITH_ERROR (ILTR_ERR_NOMEM);

//----- Remember the size of the Record buffer
gd->m_uiRecBufSize = pDataStore->m_uiInitRecBufSize;
}

//----- Allocate field buffer
gd->m_lpFldBuf = (IL_PSTR) IL_ALLOC (pDataStore->m_uiInitFldBufSize,
                                     gd->m_hFldBuf);

//----- Was field buffer allocated
if (gd->m_lpFldBuf == NULL)
    EXIT_WITH_ERROR (ILTR_ERR_NOMEM);

//----- Remember the size of the field buffer
gd->m_uiFldBufSize = pDataStore->m_uiInitFldBufSize;

//----- Do we need to allocate a conversion buffer?
if (pDataStore->m_uiConvBufSize)
{
    //----- Allocate it
    gd->m_lpConvBuf = (IL_PSTR) IL_ALLOC (pDataStore->m_uiConvBufSize,
                                         gd->m_hConvBuf);

    //----- Was conversion buffer allocated?
    if (gd->m_lpConvBuf == NULL)
        EXIT_WITH_ERROR (ILTR_ERR_NOMEM);

    //----- Remember the size of the conversion buffer
    gd->m_uiConvBufSize = pDataStore->m_uiConvBufSize;
}

//----- Are we exporting or importing?
if (ILTR_nCmd == ILTR_EXPORT)
{
    /*-----
    * If we are in ILX_V4 mode and this is actually the export before
    * import phase (Phase 10), then don't need to export if ILTR_UpdOpt
    * is set to UPD_NONE, unless the DataStore always needs to be
    * recreated on an import.
    *-----*/
    if ((ILTR_phase == ILTR_PHASE10) &&
        ((ILTR_nUpdOpt == UPD_NONE) && (!pDataStore->m_bMustRewriteData)))
        EXIT_WITH_ERROR (ILTR_ERR_NORECS);

    /*-----
    * If the "nonexistent file shortcut" is enabled, and file does not
    * exist, take the shortcut, reporting NO RECORDS.
    *-----*/
    if ( (ILTR_Flags & ILTR_FLAG_SEE_IF_FILE_EXISTS)
        && IL_DOESNT_EXIST(ILTR_szAppFile) )
        EXIT_WITH_ERROR (ILTR_ERR_NORECS);

    //----- Initialize the Application data store for reading
    iRc = OpenDataStore (tr, gd, IL_ATTR_READ);

    if (iRc == ILTR_ERR_CANCEL)
        EXIT_WITH_ERROR (iRc)
    else if (iRc)
        LOG_ERROR_AND_EXIT (iRc, iRc);

    //----- Get record count from Application; returned in gd->m_lNumRecs
    pDataStore->RecordCount (tr, gd);

    //----- Exit with error if no records to export
    if (gd->m_lNumRecs <= 0)
        EXIT_WITH_ERROR (ILTR_ERR_NORECS);

    //----- Tell ILTR how many records we have for status
    ILSetRecCount (tr, gd->m_lNumRecs);
}

```

```

//----- Inform Get processing we are going to ILIF
gd->m_nPhase = ILXT_EXPORT;

} //----- if (ILTR_nCmd == ILTR_EXPORT)

//----- We are importing (ILTR_nCmd != ILTR_EXPORT).
else
{
    //----- Load TIF database, only if operating in ILX_V3 mode.
    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    {
        //----- Create the TIF file
        if (iRc = this->CreateTIF (tr))
            LOG_ERROR_AND_EXIT (iRc, iRc);

        /*-----
        * If importing to an existing file and the reconciliation option
        * is NOT set to NONE then we must open the DataStore for reading
        * and load TIF with existing records. Also if the DataStore must
        * always be rewritten on import then we must do the Export before
        * import.
        *-----*/
        if ((!m_bNewFile) && ((ILTR_nUpdOpt != UPD_NONE) ||
            (pDataStore->m_bMustRewriteData)))
        {
            //----- Open the Application data store for reading existing data
            iRc = OpenDataStore (tr, gd, IL_ATTR_READ);

            //----- Assume zero record count if we can't open the data store
            if (iRc)
            {
                //----- User cancel is a special case
                if (iRc == ILTR_ERR_CANCEL)
                    EXIT_WITH_ERROR (iRc);

                /*-----
                * It might be good to call "ILERROR(iRc, 0)" to log the failure
                * of OpenDataStore, but it's not crucial since another
                * OpenDataStore call is coming, and we log failures there.
                *-----*/
                bOpened = FALSE;
                gd->m_lNumRecs = 0;
            }

            //----- DataStore was opened, get record count into gd->m_lNumRecs
            else // (iRc == SUCCESS -- data store successfully opened)
            {
                bOpened = TRUE;
                pDataStore->RecordCount (tr, gd);
            }

            //----- Are there any existing records?
            if (gd->m_lNumRecs > 0)
            {
                //----- Inform Get processing we are going to TIF
                gd->m_nPhase = ILXT_EXPORT_BEFORE_IMPORT;

                //----- Load the TIF file with existing data
                iRc = this->LoadTIF (tr);
                if (iRc == ILTR_ERR_CANCEL)
                    EXIT_WITH_ERROR (iRc)
                else if (iRc)
                    LOG_ERROR_AND_EXIT (iRc, iRc);
            }

            //----- Close the store, if it was opened
            if (bOpened)
            {
                iRc = pDataStore->Close (tr, gd);
                if (iRc)
                    LOG_ERROR_AND_EXIT (iRc, iRc);
            }
        } // end if (existing file)

        //----- Set the TIF reconciliation option

```

```

        iRc = ILTIFSetReconcile (tr, ILTR_nUpdOpt);
        if (iRc)
        {
            //----- Weird: bad reconcile option or TIF doesn't like us
            LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);
        }

    } //---- if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)

    //----- Now, open the data store for writing
    iRc = OpenDataStore (tr, gd, IL_ATTR_WRITE);
    if (iRc == ILTR_ERR_CANCEL)
        EXIT_WITH_ERROR (iRc)
    else if (iRc)
        LOG_ERROR_AND_EXIT (iRc, iRc);

    //----- And setup to import incoming records
    gd->m_nPhase = ILXT_IMPORT;

    } //---- if (ILTR_nCmd == ILTR_EXPORT) ... else ...

Exit:
    //----- Return status
    return iRc;

} //---- CILTranslator::Begin

/*-----
* Function    : CILTranslator::OpenDataStore
* Purpose     : Member function to open the data store, and handle password
*               failures by prompting the user for the password and retrying.
* Parameters  : tr ----- pointer to ILTR translation structure
*               gd ----- global data pointer
*               nMode --- open mode
* Returns     : Status code from open or password fetch
*-----*/
int CILTranslator::OpenDataStore
    (ILTR_PTRANSL tr,                // Pointer to translator info
     CILGLPtr    gd,                // Pointer to global info
     int         nMode)             // Open mode (read or write)
{
    //----- If building for Windows, we may need to prompt for a password
    #ifdef ILWIN
        int i;                      // Loop counter
        int iRc;                    // Return code
        ILTR_PSWD sPasswordInfo;    // Info for password prompting

        //----- Init password info structure
        IL_MEMSET (&sPasswordInfo, 0, sizeof(sPasswordInfo));
        sPasswordInfo.hWin = ILTR_hProgWin;
        sPasswordInfo.hInst = hXlatorInst;
        IL_STRCPY (sPasswordInfo.szFile, ILTR_szAppFile);
        IL_STRCPY (sPasswordInfo.szCurWD, ILTR_szCurWD);

        //----- Try opening the data store
        iRc = gd->m_pDataStore->Open (tr, gd, nMode);

        //----- Retry password up to 5 times
        for (i = 0; i < 5; i++)
        {
            //----- Handle password failure here
            if (iRc != ILTR_ERR_BADPSWD)
                break;

            //----- Get password from user
            iRc = ILGetPassword (tr, &sPasswordInfo);

            //----- Get out now if this failed
            if (iRc)
                return iRc;

            //----- Copy over result and try again
            IL_STRCPY (ILTR_szPswd, sPasswordInfo.szPswd);
            iRc = gd->m_pDataStore->Open (tr, gd, nMode);
        }
    #endif
}

```

```

    }

    return iRc;

//----- Not building for Windows, simply open the data store and return status
#else
    return gd->m_pDataStore->Open (tr, gd, nMode);
#endif

} //----- CILTranslator::OpenDataStore

/*-----
 * Function:  CILTranslator->Get member function
 * Purpose:   This routine is the implementation for the Get member function
 *            of the CILTranslator class.  It is to be used as the Get
 *            callback routine from ILTR.
 *
 * Input:     tr - pointer to ILTR translation structure
 * Return:    SUCCESS - all is well
 *            ILXT_ERR_USAGE - null pointer to global data passed
 *            ILTR_ERR_NOMEM - unable to allocate memory
 * Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
 *-----*/
int CILTranslator::Get (ILTR_PTRANSLS tr)
{
    int          iRc;          // return code variable
    int          iRc2;         // return code variable
    CILGlPtr     gd;           // pointer to application global data
    CILRecPtr     pRec;         // object for current record;
    CILDataStore * pDataStore;  // pointer to the current data store
    CILFieldArray * pFldArray;  // pointer to field array
    CILField      * pField;     // pointer to field object
    ILTR_REPEAT   sRepeatInfo;  // ILTR repeat structure
    IL_HANDLE     hExDates;     // handle to exclusion list
    ILTR_PDATES   pExDates;     // pointer to exclusion list
    INT16         nNumExDates;  // number of exclusion dates
    char          szStart       // label for start date field
                    [ILTR_MAX_FLDNAME + 1];
    char          szEnd         // label for end date field
                    [ILTR_MAX_FLDNAME + 1];

    //----- establish pointer to application global
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- fetch the DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- fetch the field array object from gd
    pFldArray = gd->m_pFldArray;

    //----- fetch the record object from gd
    pRec = gd->m_pRecord;

    //----- store pointer to repeat struct in gd
    gd->m_pRepeatInfo = &sRepeatInfo;

    //----- zero out the ILTR repeat structure
    IL_MEMSET (&sRepeatInfo, 0, sizeof (ILTR_REPEAT));

    //----- reset the repeat flag
    gd->m_bRepeatingAppt = FALSE;

    //----- Get next record to export
    if (iRc = pDataStore->BeginReadRecord (tr, gd))
    {
        //----- Check if this was a (potentially) recoverable "bad record" error
        if (iRc == ILTR_ERR_BAD_RECORD)
        {
            //----- Invalid record data, log error message (text in ILTR.RC)
            ILAppendLog ( ILTR_hLog, hXlatorInst, ILTR_MSG_BAD_RECORD,
                        ILTR_szRecName, NULL );

            //----- Skip writing this record and continue translation
            EXIT_WITH_ERROR (ILTR_SKIP_ALL);
        }
    }
}

```

```

    }

    /*-----
    * Not a "bad record error". The return code might either be a serious
    * (unrecoverable) error or simply ILTR_EOF indicating the normal end of
    * the export process. In either case, we return to export to handle it.
    *-----*/
    else
        EXIT_WITH_ERROR (iRc);
}

/*-----
* We have a valid record and all related data. Now we simply tell the
* record to process each field in accordance with the information in the
* field array
*-----*/
iRc = pRec->GetRecord (tr, gd);
if (iRc >= ILTR_SKIP_WRITE && iRc <= ILTR_SKIP_ALL)
{
    //----- Assure that EndReadRecord is called
    iRc2 = pDataStore->EndReadRecord (tr, gd);
    if (iRc2)
    {
        if (iRc2 >= ILTR_SKIP_WRITE && iRc2 <= ILTR_SKIP_ALL)
            EXIT_WITH_ERROR (iRc2);
        else
            LOG_ERROR_AND_EXIT (iRc2, iRc);
    }
    EXIT_WITH_ERROR (iRc)
}
else if (iRc)
    LOG_ERROR_AND_EXIT (iRc, iRc);

//----- reset the pointer to repeat struct in gd
gd->m_pRepeatInfo = NULL;

//----- Put out repeat structure
if (gd->m_bRepeatingAppt)
{
    //----- reset repeat flag
    gd->m_bRepeatingAppt = FALSE;

    //----- get start date field label
    if (gd->m_nStartIndex != -1)
    {
        pField = pFldArray->At (gd->m_nStartIndex);
        IL_STRCPY (szStart, pField->m_szLabel);
    }
    else
        szStart[0] = '\0';

    //----- get end date field label
    if (gd->m_nEndIndex != -1)
    {
        pField = pFldArray->At (gd->m_nEndIndex);
        IL_STRCPY (szEnd, pField->m_szLabel);
    }
    else
        szEnd[0] = '\0';

    /*-----
    * If this is a normal export then put the repeat info to ILIF
    * (or TIF) using ILPutRepeat which will make the decision.
    *-----*/
    if (gd->m_nPhase == ILXT_EXPORT)
    {
        //----- Write the repeat info to ILIF (or TIF)
        ILPutRepeat (tr, &sRepeatInfo, szStart, szEnd);

        //----- If there are exclusions, free the exclusion list.
        if (sRepeatInfo.exDates)
            IL_FREE_AND_ZERO (sRepeatInfo.hExDates, sRepeatInfo.exDates);
    }

    /*-----
    * If we are importing and running in ILX V3 mode, we are doing an

```

```

    * "export before import" and need to put the repeat information
    * directly to TIF.
    *-----*/
else if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
{
    /*-----
    * To write the repeat information to TIF we need to write TWO
    * fields. The first first is the "Basic" repeat information field
    * and the second is the list of exclusion dates which may be empty.
    *-----*/

    //----- save the exclusion list count, handle and pointer
    nNumExDates = sRepeatInfo.numExDates;
    hExDates = sRepeatInfo.hExDates;
    pExDates = sRepeatInfo.exDates;

    //----- clear handle and pointer in the Basic Repeat structure
    sRepeatInfo.hExDates = NULL;
    sRepeatInfo.exDates = NULL;

    //----- Write the Repeat basic field to TIF
    iRc = ILTIFPutField ( tr, ILTR_REP_BASIC, &sRepeatInfo,
                        sizeof (ILTR_REPEAT));
    if (iRc)
    {
        if (nNumExDates)
            IL_FREE_AND_ZERO (hExDates, pExDates);
        LOG_ERROR_AND_EXIT (iRc, iRc);
    }

    //----- Write out the exclusion list only if it is not empty
    if (nNumExDates)
    {
        iRc = ILTIFPutField ( tr, ILTR_REP_XDATE, pExDates,
                            (nNumExDates * sizeof (long)) );
        IL_FREE_AND_ZERO (hExDates, pExDates);
        if (iRc)
            LOG_ERROR_AND_EXIT (iRc, iRc);
    }
    }
}

/*-----
* Call DataStore to allow any post read record processing. This gives
* the DataStore the ability to determine if this record should actually
* be put out.
*-----*/
iRc = pDataStore->EndReadRecord (tr, gd);
if (iRc)
{
    if (iRc >= ILTR_SKIP_WRITE && iRc <= ILTR_SKIP_ALL)
        EXIT_WITH_ERROR (iRc);
    else
        LOG_ERROR_AND_EXIT (iRc, iRc);
}

//----- No errors or skip code returned, so return SUCCESS.
iRc = SUCCESS;

Exit:

//----- Set action to SKIP if necessary
if ( iRc >= ILTR_SKIP_WRITE
    && iRc <= ILTR_SKIP_ALL
    && ILTR_action == ILTR_ACT_READ )
    ILSetAction (tr, ILTR_ACT_SKIP);

//----- pass along return code, possibly an error, or a "skip" code.
return iRc;
} //---- CILTranslator::Get

/*-----
* Function: CILTranslator::Put
* Purpose: This routine is the implementation for the Put member function

```

```

*           of the CILTranslator class.  It is to be used as the Put
*           callback routine from ILTR.
*   Input:   tr - pointer to ILTR translation structure
*   Return:  SUCCESS - all is well
*           ILXT_ERR_USAGE - NULL pointer to global data passed
*           ILTR_ERR_NOMEM - unable to allocate memory
*   Author:  Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int CILTranslator::Put (ILTR_PTRANSL tr)
{
    CILGlPtr      gd;                // pointer to application global data
    ILTR_REPEAT   sRepeatInfo;       // IL Repeat info
    int           iRc;               // return code
    CILDataStore *pDataStore;        // pointer to DataStore object

    //----- establish pointer to application global
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- fetch the DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- reset repeating appointment flag
    gd->m_bRepeatingAppt = FALSE;

    //----- Check for repeat info
    sRepeatInfo.exDates = NULL;
    iRc = ILGetRepeat (tr, &sRepeatInfo);
    if (iRc && iRc != ILTR_ERR_NOFLD)
        return ILERROR (iRc, iRc);

    //----- store the pointer to the repeat info in global data
    gd->m_pRepeatInfo = &sRepeatInfo;

    /*-----
    * If no repeat data found, or if the repeat type is "no repeat" and the
    * number of days is 1, we simply put the record without repeat processing.
    *-----*/
    if ( iRc == ILTR_ERR_NOFLD ||
        (sRepeatInfo.type == ILTR_NOREPEAT && sRepeatInfo.numDays == 1) )
        iRc = PutNext (tr, &sRepeatInfo);

    /*-----
    * When doing Synchronization, we never fan here.
    * But for SmartMerge we check to see whether fanning is necessary.
    *-----*/
    else
    {
        if (ILTR_nSynchronize == ILXTR_SYNC_NO)
            iRc = pDataStore->CanPutRepeat (tr, gd);
        else
            //--- synchronizing, so fanning is not done yet
            iRc = SUCCESS;

        if (iRc == ILXT_ERR_MUST_FAN)
        {
            //----- Fan anything that we can't preserve
            iRc = ILRepeatItem (tr, &sRepeatInfo, pDataStore->m_nMaxRepeat);

            //----- Free excluded dates, if any
            if (sRepeatInfo.exDates != NULL)
                IL_FREE_MEM (sRepeatInfo.hExDates, sRepeatInfo.exDates);
        }
        else
        {
            //----- this is a repeating appointment
            gd->m_bRepeatingAppt = TRUE;

            //----- no need to fan: put out item
            iRc = PutNext (tr, &sRepeatInfo);
        }
    }

    //----- Clear the repeat information pointer and return status
    gd->m_pRepeatInfo = NULL;

```

```

    return iRc;
} //---- CILTranslator::Put

/*-----
 * Function: CILTranslator::PutNext
 * Purpose:  This routine is the implementation for the PutNext member function
 *           of the CILTranslator class. It is to be used as the Repeat
 *           callback routine from ILTR.
 *
 * NOTE:     this function is called in two different contexts:
 *
 *           * during the Sanitizing Source Records phase, also known as
 *             "Put Processing", which is used in both ILX_V3 mode, and in
 *             Phase 30 of ILX_V4 mode, ...
 *
 *           * and during the Unload-to-App phase, which is part of
 *             'End Processing'.
 *
 *           The key differentiator between these two contexts is
 *           the ILTR_nProcessStage.
 *
 * Input:    tr - pointer to ILTR translation structure
 *           pRepeat - pointer to repeat info for this rec
 * Return:    SUCCESS - all is well
 *           ILXT_ERR_USAGE - NULL pointer to global data passed
 *           ILTR_ERR_NOMEM - unable to allocate memory
 *           ILTR_ERR_CANCEL - user cancel
 *           ILXT_ERR_TIF - problems writing to TIF
 * Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
 *-----*/
int CILTranslator::PutNext (ILTR_PTRANSL tr, ILTR_PREPEAT pRepeat)
{
    int          iRc;                // return code
    int          nFields;            // number of fields in field array
    int          i;                  // loop variable
    unsigned int uiFldLen;           // length of field buffer
    unsigned int uiMaxBufSize;       // max size of field buffer
    CILGlPtr     gd;                 // pointer to application global data
    CILField     *pField;            // pointer to field object
    CILField     *pFieldRepBasic = NULL; // pointer to RepBasic field object
    CILField     *pFieldRepExcl = NULL; // pointer to RepExcl field object
    CILFieldArray *pFldArray;        // pointer field array
    CILDataStore *pDataStore;        // pointer to DataStore object
    IL_HANDLE    hExDates;           // handle to exclusion list
    ILTR_PDATES  pExDates;           // pointer to exclusion list
    INT16        nNumExDates;        // number of exclusion dates
    char         szMsg [ILTB_MAX_MSG]; // Debug message log text
    ILTR_PREPEAT pRepeatInfo;        // pointer to ILTR repeat structure

    //---- When we're called here during UNLOAD-to-APP, call UnloadInstance
    if (ILTR_nProcessStage == ILTR_END)
        return UnloadInstance (tr, pRepeat);

    //----- establish pointer to application global
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- fetch the field array from gd
    pFldArray = gd->m_pFldArray;

    //----- fetch the DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- store the pointer to the repeat info in global data
    gd->m_pRepeatInfo = pRepeat;

    //----- retrieve number of fields.
    nFields = pFldArray->m_nFields;

    //----- retrieve max buffer size from data store
    uiMaxBufSize = pDataStore->m_uiMaxFldBufSize;

    /*-----

```



```

* Add the current IntelliLink ILIF record to the TIF reconciliation
* database. TIF will handle detecting and resolving conflicts.
*-----*/

//----- Place a new line in the debug log if debug logging.
IL_OutputDebugLog (tr, "");
IL_OutputDebugLog (tr, "Importing Record");

//----- Put each field in the IntelliLink field list to the TIF DB
for (i = 0; i < nFields; ++i)
{
    //----- get current field object from field array
    pField = pFldArray->At (i);

    //----- skip empty slots in field array
    if (pField == NULL)
        continue;

    //----- Skip the repeat fields; we handle them last.
    if (pField->m_nFieldAction == ILXT_ACT_REPEATBASIC)
    {
        pFieldRepBasic = pField;
        continue;
    }

    if (pField->m_nFieldAction == ILXT_ACT_REPEATXDATES)
    {
        pFieldRepExcl = pField;
        continue;
    }

    /*-----
    * Skip ID fields and the _subType and _Delta fields.
    * If the SST mechanism is enabled then the _subType field is put into
    * TIF when ILTIFPutField is called for the TAGGED field.
    *-----*/
    if ( pField->m_nFieldAction == ILXT_ACT_RECORDID ||
        pField->m_nFieldAction == ILXT_ACT_UNIQUEID ||
        pField->m_nFieldAction == ILXT_ACT_DELTA )
        continue;

    //----- skip (non-hidden) unmapped fields?
    if ( pDataStore->m_bOnlyMappedInTIF &&
        pField->m_bIsMapped == FALSE &&
        !(pField->m_ulAttribs & ILTB_ATT_HIDDEN_FIELD) )
        continue;

    //----- Get field data, grow field buffer as needed, up to maximum
    uiFldLen = gd->m_uiFldBufSize;
    while (ILTR_ERR_TRUNC == (iRc = ILFldGet (tr, pField->m_szLabel,
        gd->m_lpFldBuf, &uiFldLen)))
    {
        //----- Report (non-fatal) error if buffer already at maximum size
        if (gd->m_uiFldBufSize >= uiMaxBufSize)
        {
            ILAddFieldError (tr, pField->m_szLabel, ILTR_ERR_TRUNC);
            break;
        }

        //----- Otherwise, grow the field buffer and try again
        else
        {
            //----- increment buffer size
            gd->m_uiFldBufSize += pDataStore->m_uiBufIncSize;

            //----- is it too big?
            if (gd->m_uiFldBufSize > uiMaxBufSize)
                gd->m_uiFldBufSize = uiMaxBufSize;

            //----- save off new field buffer size
            uiFldLen = gd->m_uiFldBufSize;

            //----- reallocate the buffer
            gd->m_lpFldBuf = (IL_PSTR) IL_REALLOC ( gd->m_uiFldBufSize,
                gd->m_hFldBuf,
                gd->m_lpFldBuf );
        }
    }
}

```

```

        //----- was reallocation successful?
        if (! gd->m_lpFldBuf)
            EXIT_WITH_ERROR (ILTR_ERR_NOMEM);
    }
}

//----- Check for other errors which ILFldGet may return
if (iRc && (iRc != ILTR_ERR_NODATA)
    && (iRc != ILTR_ERR_NOTMAPPED)
    && (iRc != ILTR_ERR_NOFLD))
{
    ILAddFieldError (tr, pField->m_szLabel, iRc);
    if (iRc == ILTR_ERR_NOMEM || iRc == ILTR_ERR_FILE)
    {
        ILSetAction (tr, ILTR_ACT_IGNORE);
        EXIT_WITH_ERROR (iRc);
    }
}

//----- If we have valid field data, put it to the TIF DB record now
if (uiFldLen)
{
    //----- Don't truncate values here; let CILRecord::PutRecord do that.
    //----- TIF is smart enough to make truncated values compare equal.

    //----- Cleanse the field value
    iRc = pDataStore->CleanseField ( tr, gd, pField, gd->m_lpFldBuf,
                                     &uiFldLen, // current len (IN & OUT)
                                     gd->m_uiFldBufSize ); // max length

    if (iRc)
        ILERROR (iRc, ILTR_ERR_ILXTRANS_STORE);

    iRc = ILTIFPutField (tr, pField->m_szLabel, gd->m_lpFldBuf, uiFldLen);
    if (iRc)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

    //----- Make up message for log file.
    MakeLogMessage (tr, szMsg, pField, gd->m_lpFldBuf);
    IL_OutputDebugLog (tr, szMsg);
}

//----- Put out repeat info if we have a usable ILTR_REPEAT structure
if (gd->m_bRepeatingAppt && pFieldRepBasic != NULL)
{
    //----- Get a pointer to the repeat structure and reset repeat flag
    pRepeatInfo = gd->m_pRepeatInfo;
    gd->m_bRepeatingAppt = FALSE;

    /*-----
    * To write the repeat information to TIF we need to write TWO
    * fields. The first first is the "Basic" repeat information field
    * and the second is the list of exclusion dates which may be empty.
    *-----*/

    //----- save the exclusion list count, handle and pointer
    nNumExDates = pRepeatInfo->numExDates;
    hExDates = pRepeatInfo->hExDates;
    pExDates = pRepeatInfo->exDates;

    //----- clear handle and pointer in the Basic Repeat structure
    pRepeatInfo->hExDates = NULL;
    pRepeatInfo->exDates = NULL;

    //----- Cleanse the field value
    uiFldLen = sizeof (ILTR_REPEAT);
    iRc = pDataStore->CleanseField ( tr, gd, pFieldRepBasic,
                                     (IL_PSTR) pRepeatInfo,
                                     &uiFldLen, // current len (IN & OUT)
                                     uiFldLen ); // maximum length

    if (iRc)
        ILERROR (iRc, ILTR_ERR_ILXTRANS_STORE);

    //----- Write the Repeat basic field to TIF
    iRc = ILTIFPutField (tr, ILTR_REP_BASIC, pRepeatInfo, uiFldLen);
}

```

```

    if (iRc) ILERROR (iRc, iRc);

    //----- Write out the exclusion list only if it is not empty
    if (iRc == SUCCESS && nNumExDates > 0 && pFieldRepExcl != NULL)
    {
        //----- Cleanse the field value
        uiFldLen = nNumExDates * sizeof (long);
        iRc = pDataStore->CleanseField ( tr, gd, pFieldRepExcl,
                                         (IL_PSTR) pExDates,
                                         &uiFldLen, // cur len (IN & OUT)
                                         uiFldLen ); // maximum length

        if (iRc)
            ILERROR (iRc, ILTR_ERR_ILXTRANS_STORE);

        iRc = ILTIFPutField (tr, ILTR_REP_XDATE, pExDates, uiFldLen);
        if (iRc) ILERROR (iRc, iRc);
    }

    if (nNumExDates > 0)
        IL_FREE_AND_ZERO (hExDates, pExDates);

    if (iRc != SUCCESS)
        EXIT_WITH_ERROR (iRc);
}

//----- All fields in the field list processed, write the TIF DB record
if (iRc = ILTIFWriteRecord (tr))
{
    //----- If user cancelled, we need to remember this for end processing
    if (iRc == ILTR_ERR_CANCEL)
    {
        gd->m_bUserCancel = TRUE;
        EXIT_WITH_ERROR (ILTR_ERR_CANCEL);
    }

    //----- ignore TIF_RECERROR, caused by writing record with no data
    if ((iRc) && (iRc != TIF_RECERROR))
        //----- Treat all other errors as serious TIF errors
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);
}

//----- all is well
iRc = SUCCESS;

Exit:
    //----- all done
    return iRc;
} //---- CILTranslator::PutNext

/*-----
* Function: CILTranslator::End
* Purpose: This routine is the implementation for the End member function
* of the CILTranslator class. It is to be used as the End
* callback routine from ILTR.
* Input: tr - pointer to ILTR translation structure
* Return: SUCCESS - all is well
* ILXT_ERR_USAGE - NULL pointer to global data passed
* ILTR_ERR_NOMEM - unable to allocate memory
* Author: Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int CILTranslator::End (ILTR_PTRANSL tr)
{
    int iRc = SUCCESS; // return code
    int iRc2 = SUCCESS; // secondary return code
    CILGLPtr gd; // pointer to application global data
    CILDataStore *pDataStore; // pointer to DataStore object
    CILFieldArray *pFldArray; // pointer to FieldArray object

    //----- establish pointer to application global
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- fetch the DataStore object from gd

```

```

pDataStore = gd->m_pDataStore;

//----- fetch the FieldArray object from gd
pFldArray = gd->m_pFldArray;

/*-----
 * If the DataStore or FieldArray pointers are NULL, we assume that
 * Begin processing was never completed, so we simply return SUCCESS.
 *-----*/
if (pDataStore == NULL || pFldArray == NULL)
    EXIT_WITH_ERROR (SUCCESS);

//----- if we are importing and user hasn't cancelled - unload TIF file
if ((ILTR_direction == ILTR_IMPORT) && (!gd->m_bUserCancel) &&
    (ILTR_rc == SUCCESS))
{
    //----- If Importing and "selected" is set, invoke the Chooser
    if (ILTR_Flags & ILTR_FLAG_IMPORT_SELECTED)
    {
        if ( (ILTR_phase == 0) ||
            (ILTR_phase == ILTR_PHASE30 && ILTR_nSynchronize == 0) )
        {
            //----- Call the Chooser dialog to select records to be processed
            iRc = CallChooseRecords (tr);

            //----- Remap IDOK and IDCANCEL return codes from Chooser dialog
            if (iRc == IDOK)
                iRc = SUCCESS;
            else if (iRc == IDCANCEL)
                iRc = ILTR_ERR_CANCEL;

            //----- Check for error condition (or user cancel)
            if (iRc)
                EXIT_WITH_ERROR (iRc)
        }
    }

    //----- we are done loading tif, time to unload
    if (ILTR_phase != ILTR_PHASE40)
    {
        //--- But don't call ILTIFEndLoad more than once. We call it before
        //--- unloading to TARGET, but not before unloading to SOURCE.
        iRc = ILTIFEndLoad (tr);
        if (iRc == ILTR_ERR_CANCEL)
            EXIT_WITH_ERROR (iRc)
        else if (iRc != SUCCESS)
            LOG_ERROR_AND_EXIT (iRc, iRc);
    }

    /*-----
     * For Synchronization Only, if target system needs to have all fanning
     * done before we start unloading records from TIF then call the
     * FanBeforeUnload function to scan all records in TIF looking for any
     * that need to be fanned, and creating fanned instances in TIF
     * for those that do need to be fanned.
     *-----*/
    if ( (ILTR_nAttribs & ILTB_ATT_MUST_FAN_B4U)
        && (ILTR_nSynchronize != ILXTR_SYNC_NO)
        && (ILTR_nFunction == ILTR_APPT || ILTR_nFunction == ILTR_TODO) )
    {
        iRc = this->FanBeforeUnload (tr);
        if (iRc == ILTR_ERR_CANCEL)
            EXIT_WITH_ERROR (iRc)
        else if (iRc != SUCCESS)
            LOG_ERROR_AND_EXIT (iRc, iRc);
    }

    //----- Unload the records in TIF
    iRc = this->UnloadTIF (tr);
    if (iRc == ILTR_ERR_CANCEL)
        EXIT_WITH_ERROR (iRc)
    else if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, iRc);
}

//----- all is well

```

```

    iRc = SUCCESS;

Exit:
    /*-----
    * Cleanup
    *-----*/

    //----- Close the TIF file if in import mode
    if (ILTR_direction == ILTR_IMPORT && pFldArray)
    {
        //----- Close and delete the TIF file if doing an ILX_V3 Import
        if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
        {
            iRc2 = ILTIFClose (tr);
            if (iRc2 != SUCCESS)
                iRc2 = ILERROR (iRc2, ILXT_ERR_TIF);
        }
    }

    if (iRc == SUCCESS)
        //--- no previous error, so pass back this error
        iRc = iRc2;

    /*-----
    * Update ILTR_rc so that data store close can be sensitive to whether
    * translation has succeeded or failed.
    *-----*/
    if (ILTR_rc == SUCCESS)
        ILTR_rc = iRc;

    //----- close the data store if it has been successfully created.
    if (pDataStore)
    {
        iRc2 = pDataStore->Close (tr, gd);
        if (iRc2 != SUCCESS)
            iRc = iRc2;
    }

    //----- Free the record buffer
    if (gd->m_lpRecBuf)
        IL_FREE_AND_ZERO (gd->m_hRecBuf, gd->m_lpRecBuf);

    //----- Free the field buffer
    if (gd->m_lpFldBuf)
        IL_FREE_AND_ZERO (gd->m_hFldBuf, gd->m_lpFldBuf);

    //----- Free the conversion buffer
    if (gd->m_lpConvBuf)
        IL_FREE_AND_ZERO (gd->m_hConvBuf, gd->m_lpConvBuf);

    //----- all done
    return iRc;
} //---- CILTranslator::End

/*-----
* Function:  CILTranslator::New
* Purpose:   This routine is the implementation for the New member function
*            of the CILTranslator class. It is to be used as the New
*            callback routine from ILTR.
* Input:     tr - pointer to ILTR translation structure
* Return:    SUCCESS - all is well
*            ILXT_ERR_USAGE - NULL pointer to global data passed
* Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int CILTranslator::New
    (ILTR_PTRANSL tr)          // pointer to ILTR translator structure
{
    int          iRc;          // return code
    CILGlPtr     gd;           // pointer to GlobalData object
    CILDataStore *pDataStore;  // pointer to DataStore object

    //----- establish pointer to application global
    gd = this->m_pGlobalData;
    if (gd == NULL)

```

```

        return ILERROR (0, ILXT_ERR_USAGE);

//----- remember that we are processing a new file
m_bNewFile = TRUE;

//----- fetch the datastore object from gd
pDataStore = gd->m_pDataStore;

//----- make sure the DataStore object has been created
if (pDataStore == NULL)
    return ILERROR (0, ILXT_ERR_USAGE);

//----- tell the DataStore to create new data repository (file, etc..)
iRc = pDataStore->New (tr, gd);

//----- pass on result
if (iRc == SUCCESS || iRc == ILTR_ERR_CANCEL)
    return iRc;
else
    return ILERROR (iRc, iRc);
} //----- CILTranslator::New

/*-----
* Function:    CILTranslator::CreateTIF
* Purpose:    Create an empty translator intermediate file, define TIFfields
*             based on fields in the FieldArray object.
* Parameters: tr - Pointer to translation information
* Returns:    SUCCESS - All is well
*             ILXT_ERR_USAGE - NULL pointer to global data is passed
*             ILTR_ERR_BADMAP - Something seriously wrong with field map
*             ILXT_ERR_TIF - Unable to access TIF file for some reason
* Notes:
*   This function is called once at the beginning of each data
*   section to create a Translate Intermediate File (TIF).
*   The TIF file is used to temporarily store all data records
*   for reconciliation purposes.
*-----*/
int CILTranslator::CreateTIF
    (ILTR_PTRANSL tr)          // Pointer to ILTR translator struct
{
    int          i;              // Loop variable
    int          iRc = SUCCESS;  // Return code
    long         lFields = 0;    // Count of mapped fields
    CILFieldArray *pFldArray;    // ptr to FieldArray object
    CILDataStore *pDataStore;    // ptr to DataStore object
    CILField     *pField;        // ptr to Field Object
    CILGlPtr     gd;             // ptr to global data object

//----- set up gd to point to global data
gd = this->m_pGlobalData;
if (gd == NULL)
    return ILERROR (0, ILXT_ERR_USAGE);

//----- Fetch the fieldArray object from gd
pFldArray = gd->m_pFldArray;

//----- Fetch the DataStore object from gd
pDataStore = gd->m_pDataStore;

//----- Should just mapped/hidden fields go in TIF?
if (pDataStore->m_bOnlyMappedInTIF)
{
    //----- Walk the list of fields counting mapped or hidden fields.
    for (i = 0; i < pFldArray->m_nFields; i++)
    {
        //----- index into field array
        pField = pFldArray->At (i);
        if (!pField)
            continue;

        //----- count number of mapped and/or hidden fields
        if ( (pField->m_bIsMapped) ||
            (pField->m_ulAttribs & ILTB_ATT_HIDDEN_FIELD) )
            lFields++;
    }
}
}

```

```

    }
}
else //----- all fields go in TIF
{
    lFields = pFldArray->m_nFields;
}

//----- Create the TIF database with the appropriate number of fields.
iRc = ILTIFCreate (tr, hXlatorInst, lFields);
if (iRc != SUCCESS)
    LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

/*-----
 * Walk the list of fields once more to create field descriptors in TIF.
 *-----*/
for (i = 0; i < pFldArray->m_nFields; i++)
{
    //----- get current field object
    pField = pFldArray->m_pFields [i];
    if (!pField)
        continue;

    //----- skip unmapped (non-hidden) fields?
    if ( pDataStore->m_bOnlyMappedInTIF  &&
        pField->m_bIsMapped == FALSE    &&
        !(pField->m_ulAttribs & ILTB_ATT_HIDDEN_FIELD) )
        continue;

    //----- Provide TIF field descriptor
    iRc = ILTIFDefFieldN (tr, pField->m_nILTRIndex, 0);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);
}

Exit:
//----- All done, return status.
return (iRc);

} //----- CILTranslator::CreateTIF

/*-----
 * Name:          CILTranslator::LoadTIF
 * Purpose:       Load all existing data into the translator intermediate file
 * Parameters:    tr - Pointer to ILTR translation information
 *               gd - Pointer to GlobalData object
 * Returns:       SUCCESS - All is well
 *               ILXT_ERR_USAGE - NULL pointer to global data passed
 *               ILXT_ERR_TIF - Unable to put record into TIF
 * Notes:
 *-----*/
int CILTranslator::LoadTIF
    (ILTR_PTRANSL tr)          // pointer to ILTR translation info
{
    int      iRc = SUCCESS;      // Return code
    CILGlPtr gd;                // ptr to global data object
    char      szMsg [ILTB_MAX_MSG + 1]; // Message buffer

    //----- set up gd to point to global data
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- Set up progress bar for building the reconciliation database
    LoadString ((IL_HINST) hXlatorInst, ILXT_RSRC_LOADMSG, szMsg,
        ILTB_MAX_MSG);
    ILStatusInit (tr, szMsg, gd->m_lNumRecs);

    //----- Log debug message at start of loading TIF database.
    IL_OutputDebugLog (tr, "");
    IL_OutputDebugLog (tr, "Preloading TIF database");

    //----- Load each existing record into TIF until ILTR_EOF reached.
    while (TRUE)
    {
        //----- Invoke standard get processing, until EOF is reached.

```

```

        iRc = this->Get (tr);
        if (iRc == ILTR_EOF)
            break;

        //----- EOF not reached yet, did we get a skip or error code?
        if (iRc != SUCCESS)
        {
            //----- End loop and return status if this is not a "skip code".
            if (iRc < 0 || iRc > ILTR_SKIP_ALL)
                break;

            //----- Skip writing this record to TIF on ILTR_SKIP_WRITE.
            if (iRc & ILTR_SKIP_WRITE)
                continue;
        }

        //----- All fields in the field list processed, write the TIF DB record
        if (iRc == ILTIFWriteRecord (tr))
        {
            //----- If user cancelled, we need to remember this for end processing
            if (iRc == ILTR_ERR_CANCEL)
            {
                gd->m_bUserCancel = TRUE;
                return ILTR_ERR_CANCEL;
            }

            //----- ignore TIF_RECERROR, caused by writing record with no data
            if ((iRc) && (iRc != TIF_RECERROR))
            {
                //----- Treat all other errors as serious TIF errors
                return ILERROR (iRc, ILXT_ERR_TIF);
            }
        }

        //----- Update the progress bar for each record processed
        ILStatusUpdate (tr);

        //----- Process messages and check for user cancel
        if (iRc == ILProcessMessages (tr))
        {
            if (iRc == ILTR_ERR_CANCEL)
                gd->m_bUserCancel = TRUE;
            return iRc;
        }
    }

    //----- Show all records copied from the export file to the TIF database
    ILStatusDone (tr);

    //----- If Data store returned ILTR_EOF, treat it as SUCCESS.
    if (iRc == ILTR_EOF)
        iRc = SUCCESS;

    //----- Log debug message at end of loading TIF database.
    IL_OutputDebugLog (tr, "");
    IL_OutputDebugLog (tr, "TIF database loaded. Translation begins.");

    //----- All done
    return iRc;
} //----- CILTranslator::LoadTIF

/*-----
* Function:    CILTranslator::UnloadTIF
* Purpose:    Unload all TIF data into the current data store
* Parameters: tr - Pointer to translation information
* Returns:    SUCCESS - All is well
*             ILXT_ERR_USAGE - NULL pointer to global data passed
*             ILXT_ERR_TIF - abnormal TIF error
* Notes:
*-----*/
int CILTranslator::UnloadTIF
    (ILTR_PTRANS tr)                // pointer to ILTR translation info
{
    int         iRc = SUCCESS;        // Return code
    LONG        lRecords;             // Number of TIF records
    CILGLPtr    gd;                  // ptr to GlobalData object

```



```

char          szMsg [ILTB_MAX_MSG + 1];    // Message buffer

//----- Set up gd to point to Global data object
gd = this->m_pGlobalData;
if (gd == NULL)
    return ILERROR (0, ILXT_ERR_USAGE);

//----- Retrieve number of records from TIF
iRc = ILTIFHowManyRecords (tr, &lRecords);
if (iRc != SUCCESS)
    return ILERROR (iRc, ILXT_ERR_TIF);

gd->m_lNumRecs = lRecords;

//----- Initialize progress bar
LoadString ((IL_HINST) hXlatorInst, ILXT_RSRC_UNLOADMSG, szMsg,
            ILTB_MAX_MSG);
ILStatusInit (tr, szMsg, lRecords);

//----- Retrieve all records from TIF
do
{
    //----- Process each record in TIF
    if (iRc = this->UnloadRecord (tr))
        break;

    //----- Update the progress bar for each record processed
    ILStatusUpdate (tr);

    //----- Process messages and check for user cancel
    if (iRc = ILProcessMessages (tr))
    {
        if (iRc == ILTR_ERR_CANCEL)
            gd->m_bUserCancel = TRUE;
        return iRc;
    }
}
while ((iRc = ILTIFNextRecord (tr)) == SUCCESS);

//----- Did we reach normal end of file?
if (iRc == TIF_EOF)
    iRc = SUCCESS;

//----- Show all records copied from the export file to the TIF database
ILStatusDone (tr);

//----- All done
if (iRc == SUCCESS || iRc == ILTR_ERR_CANCEL)
    return iRc;
else
    return ILERROR (iRc, iRc);
} //----- CILTranslator::UnloadTIF

/*-----
* Function:    CILTranslator::UnloadRecord
* Purpose:     Unload a TIF record to data store
* Parameters:  tr - Pointer to translation information
* Returns:     SUCCESS - All is well
*             ILXT_ERR_USAGE - NULL pointer to global data passed
* Notes:
*-----*/
int CILTranslator::UnloadRecord
    (ILTR_PTRANSI tr)          // pointer to ILTR translation info
{
    int          iRc = SUCCESS;          // Return code
    int          iRc2;                   // Secondary return code
    INT32        lTIFOutcome;            // TIF "outcome" of this record
    IL_PSTR      pUniqueID;              // Pointer to NEW unique ID
    CILGlPtr     gd;                     // ptr to GlobalData object
    CILRecPtr     pRec;                  // object for current record;
    ILTR_REPEAT  sRepeatInfo;            // repeat struct for this record

    //----- Set up gd to point to Global data object

```

```

gd = this->m_pGlobalData;
if (gd == NULL)
    return ILERROR (0, ILXT_ERR_USAGE);

//----- store pointer to repeat struct in gd
IL_MEMSET (&sRepeatInfo, '\0', sizeof (sRepeatInfo));
gd->m_pRepeatInfo = &sRepeatInfo;

//----- reset repeating appointment flag
gd->m_bRepeatingAppt = FALSE;

//----- fetch the record object from gd
pRec = gd->m_pRecord;

//----- Read next TIF record into buffer
iRc = ILTIFReadRecord (tr);
if (iRc == TIF_EOF)
{
    EXIT_WITH_ERROR (TIF_EOF);
}
else if (iRc != SUCCESS)
    LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

//----- Get synchronization or conflict resolution outcome for this record
iRc = ILTIFGetOutcome (tr, &lTIFOutcome);
if (iRc != SUCCESS)
    LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

//----- Determine if record should be added, changed, or replaced
if (lTIFOutcome & ILTIF_OUTCOME_ADD)
    gd->m_nRecAction = UPD_INSERT;
else if (lTIFOutcome & ILTIF_OUTCOME_UPDATE)
    gd->m_nRecAction = UPD_UPDATE;
else if (lTIFOutcome & ILTIF_OUTCOME_REPLACE)
    gd->m_nRecAction = UPD_REPLACE;
else if (lTIFOutcome & ILTIF_OUTCOME_DELETE)
    gd->m_nRecAction = UPD_DELETE;
else if (lTIFOutcome & ILTIF_OUTCOME_DELTA_ACK)
    gd->m_nRecAction = UPD_DELTA_ACK;
else if (lTIFOutcome & ILTIF_OUTCOME_LEAVE_ALONE)
    gd->m_nRecAction = UPD_NONE;
else
    LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_UNK_VAL);

//----- Reset the record and unique ID values
gd->m_ulRecID = (UINT32) -1L;
gd->m_szRecID[0] = '\0';
gd->m_szUniqueID[0] = '\0';

//----- Reset pointer to unique ID to be passed to ILTIFAcceptOutcome.
pUniqueID = NULL;

//----- based on action code, call appropriate record member function
switch (gd->m_nRecAction)
{
    case UPD_INSERT:                // add a new record

        //--- For Synchronization Only, determine whether FANNING is required
        if ( (ILTR_nSynchronize != ILXTR_SYNC_NO)
            && (ILTR_nFunction == ILTR_APPT || ILTR_nFunction == ILTR_TODO) )
        {
            BOOLEAN bFannedHere = FALSE;
            iRc = ILGetRepeat (tr, gd->m_pRepeatInfo);
            if (iRc && iRc != ILTR_ERR_NOFLD)
                return ILERROR (iRc, iRc);

            /*-----
            * If repeat pattern is non-trivial, fanning may be required...
            *-----*/
            if ( (iRc != ILTR_ERR_NOFLD)
                && ( gd->m_pRepeatInfo->type != ILTR_NOREPEAT
                    || gd->m_pRepeatInfo->numDays != 1 ) )
            {
                iRc = gd->m_pDataStore->CanPutRepeat (tr, gd);
                if (iRc == ILXT_ERR_MUST_FAN)
                {

```

```

        bFannedHere = TRUE;

        //----- Fan out this recurring item into instances...
        iRc = ILRepeatItem ( tr, gd->m_pRepeatInfo,
                            gd->m_pDataStore->m_nMaxRepeat );
    }

    /*-----
    * Free exclusion list (NOTE: the CILField::ProcessPut function,
    * in CILFIELD.CPP, will call ILGetRepeat again, into the same
    * buffer that we use here, so we have to free the exclusion list
    * here to avoid creating a memory leak.)
    *-----*/
    if (gd->m_pRepeatInfo->exDates != NULL)
        IL_FREE_AND_ZERO ( gd->m_pRepeatInfo->hExDates,
                           gd->m_pRepeatInfo->exDates );
    if (bFannedHere)
    {
        gd->m_bRepeatingAppt = FALSE;
        return iRc;
    }
    //.....no fanning required, drop into next case to Put Record...

case UPD_REPLACE:           // replace an existing record
case UPD_UPDATE:           // update existing record

    //----- Add, replace, or update a record
    iRc = pRec->PutRecord (tr, gd);

    //----- If there is a new Unique ID, set a pointer to this ID
    if (gd->m_bHasUniqueID && gd->m_szUniqueID[0])
        pUniqueID = gd->m_szUniqueID;
    break;

case UPD_NONE:              // no change to an existing record

    //----- Process unchanged record
    //----- NOTE: TIF does not allow a new Unique ID to be assigned here.
    //----- It seems unlikely that anyone will ever want to assign a new
    //----- ID here, but if it happens TIF will need a small tweak.
    iRc = pRec->PutRecord (tr, gd);
    break;

case UPD_DELETE:           // delete the existing record

    //----- Delete the record
    iRc = pRec->Delete (tr, gd);
    break;

case UPD_DELTA_ACK:        // acknowledge FastSync Delta

    //----- Delete the record
    iRc = pRec->FastSyncDeltaAck (tr, gd);
    break;

default:                   // should never get here

    iRc = ILERROR (gd->m_nRecAction, ILXT_ERR_UNK_VAL);
    break;
}

Exit:

/*-----
* Did we succeed in processing the record? If so, then we tell
* TIF that all is well. Otherwise, we tell TIF to reject the
* record. TIF will take care of reflecting the appropriate record
* status in the log file.
*-----*/
if (iRc == SUCCESS)
    iRc = ILTIFAcceptOutcome (tr, pUniqueID);
else if (iRc != TIF_EOF)
    iRc2 = ILTIFRejectOutcome (tr, iRc);

```

```

//----- Clear SKIP messages
if (iRc >= ILTR_SKIP_WRITE && iRc <= ILTR_SKIP_ALL)
    iRc = SUCCESS;

//----- free memory used by repeat exclusion list
//----- (allocated by CILField::ProcessPut, in CILFIELD.CPP)
if (gd->m_bRepeatingAppt)
{
    if (sRepeatInfo.hExDates)
        IL_FREE_AND_ZERO (sRepeatInfo.hExDates, sRepeatInfo.exDates);
    gd->m_bRepeatingAppt = FALSE;
}

//----- Reset pointer to repeat struct in gd
gd->m_pRepeatInfo = NULL;

//----- All done
if (iRc == SUCCESS || iRc == TIF_EOF || iRc == ILTR_ERR_CANCEL)
    return iRc;
else
    return ILERROR (iRc, iRc);
} //----- CILTranslator::UnloadRecord

/*-----
* Function: CILTranslator::UnloadInstance
* Purpose: This routine is the implementation for the UnloadInstance
* member function of the CILTranslator class. It is called from
* CILTranslator::PutNext, which is called from the
* Repeat Callback routine (*ILTR_cbRepeat) while we are unloading
* records from TIF. If a record that we're unloading needs to
* be fanned then CILTranslator::UnloadRecord calls ILRepeatItem,
* which in turn calls this function repeatedly, once for each
* instance of the recurring item.
*
* NOTE: this function is called for outcome=ADD (insert) ONLY!!
*
* Input: tr - pointer to ILTR translation structure
* pRepeat - pointer to repeat info for this rec
* Return: SUCCESS - all is well
* ILXT_ERR_USAGE - NULL pointer to global data passed
* ILTR_ERR_NOMEM - unable to allocate memory
* ILTR_ERR_CANCEL - user cancel
* ILXT_ERR_TIF - problems writing to TIF
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int CILTranslator::UnloadInstance (ILTR_PTRANSL tr, ILTR_PREPEAT pRepeat)
{
    int iRc; // return code
    int iRc2; // Secondary return code
    IL_PSTR pUniqueID; // Pointer to NEW unique ID
    CILGlPtr gd; // ptr to GlobalData object
    CILRecPtr pRec; // object for current record;

    //----- Set up gd to point to Global data object
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- fetch the record object from gd
    pRec = gd->m_pRecord;

    //----- Reset the record and unique ID values
    gd->m_ulRecID = (UINT32) -1L;
    gd->m_szRecID[0] = '\0';
    gd->m_szUniqueID[0] = '\0';

    //----- Reset pointer to unique ID to be passed to ILTIFAcceptOutcome.
    pUniqueID = NULL;

    //----- Add or replace a record
    iRc = pRec->PutRecord (tr, gd);

    //----- If there is a new Unique ID, set a pointer to this ID
    if (gd->m_bHasUniqueID && gd->m_szUniqueID[0])

```

```

    pUniqueID = gd->m_szUniqueID;

    //----- If record action processed successfully, "accept" the TIF outcome.
    if (iRc == SUCCESS)
        iRc = ILTIFAcceptOutcome (tr, pUniqueID);

    //----- If we have an error, we tell TIF we must "reject" the outcome
    else
        iRc2 = ILTIFRejectOutcome (tr, iRc); // iRc2 is ignored

    //----- Clear SKIP return codes
    if (iRc >= ILTR_SKIP_WRITE && iRc <= ILTR_SKIP_ALL)
        iRc = SUCCESS;

    if (iRc == SUCCESS || iRc == ILTR_ERR_CANCEL)
        return iRc;
    else
        return ILERROR (iRc, iRc);
} //---- CILTranslator::UnloadInstance

/*-----
* Function:  CILTranslator::FanBeforeUnload
* Purpose:   This function creates fanned instances, in TIF, for all
*            recurring items that need to be fanned.
*
* Method:    The TIF database is scanned, looking for recurring
*            items that are to be ADDED (inserted) to the Target App.
*            For each such item the Target App data store's
*            "CanPutRepeat" function is called to find out whether
*            the item must be fanned.  If the item must be fanned
*            then the ILTIFFanItem function is called to do the fanning.
*
* Input:     tr - pointer to ILTR translation structure
*
* Return:    SUCCESS - all is well
*            ILTR_ERR_CANCEL if user presses CANCEL
*            or ILXT_ERR_TIF or other abnormal error indication
*
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int CILTranslator::FanBeforeUnload (ILTR_PTRANSL tr)
{
    int          iRc;                // return code
    LONG          RecordCount;       // Number of TIF records
    CILGlPtr      gd;                // ptr to GlobalData object
    INT32         recnum;             // TIF record number
    INT32         Outcome;           // outcome for TIF record
    ILTR_REPEAT   sRepeatInfo;       // repeat struct for a record

    //----- Set up gd to point to Global data object
    gd = this->m_pGlobalData;
    if (gd == NULL)
        return ILERROR (0, ILXT_ERR_USAGE);

    //----- tell TIF that we're going to do 'Fanning Before Unload'
    iRc = ILTIFStartNextPhase (tr, TIF_PHASE_FANNING_BEFORE_UNLOAD);
    if (iRc != SUCCESS)
        return ILERROR (iRc, ILXT_ERR_TIF);

    //----- provide space for reading repeat structures into
    gd->m_pRepeatInfo = &sRepeatInfo;

    //----- Retrieve number of records from TIF
    iRc = ILTIFFHowManyRecords (tr, &RecordCount);
    if (iRc != SUCCESS)
        return ILERROR (iRc, ILXT_ERR_TIF);

    for (recnum=0; recnum < RecordCount; recnum++)
    {
        //----- Process messages and check for user cancel
        iRc = ILProcessMessages (tr);
        if (iRc == ILTR_ERR_CANCEL)
            gd->m_bUserCancel = TRUE;
    }
}

```

```

    if (iRc != SUCCESS)
        return iRc; // CANCEL or abnormal error

    iRc = ILTIFNextRecord (tr);
    if (iRc != SUCCESS)
        return ILERROR (iRc, ILXT_ERR_TIF);

    iRc = ILTIFGetOutcome (tr, &Outcome);
    if (iRc != SUCCESS)
        return ILERROR (iRc, ILXT_ERR_TIF);

    //---- if outcome isn't ADD or UPDATE we don't do fanning here.
    if ((Outcome & (ILTIF_OUTCOME_ADD | ILTIF_OUTCOME_UPDATE)) == 0)
        continue;

    iRc = ILTRTIFItemIsRecurring (tr);
    if (iRc == FALSE)
        //--- if item isn't recurring we obviously won't fan it.
        continue;

    else if (iRc != TRUE)
        return ILERROR (iRc, ILXT_ERR_TIF);

    //----- Read next TIF record into buffer
    iRc = ILTIFReadRecord (tr);
    if (iRc != SUCCESS)
        return ILERROR (iRc, ILXT_ERR_TIF);

    iRc = ILGetRepeat (tr, gd->m_pRepeatInfo);
    if (iRc == ILTR_ERR_NOFLD)
        continue;

    else if (iRc != SUCCESS)
        return ILERROR (iRc, iRc);

    //----- Determine whether item can be put into Target App w/o Fanning
    iRc = gd->m_pDataStore->CanPutRepeat (tr, gd);

    //----- Free excluded dates, if any
    if (gd->m_pRepeatInfo->exDates != NULL)
        IL_FREE_AND_ZERO (gd->m_pRepeatInfo->hExDates, gd->m_pRepeatInfo->exDates);

    if (iRc == ILXT_ERR_MUST_FAN)
    {
        iRc = ILTRTIFFanItem (tr, gd->m_pDataStore->m_nMaxRepeat);
        if (iRc != SUCCESS)
            return ILERROR (iRc, ILXT_ERR_TIF);
    }

    //---- tell TIF that we're finished 'Fanning Before Unload'
    iRc = ILTIFStartNextPhase (tr, TIF_PHASE_FINISHED_FANNING_BEFORE_UNLOAD);
    if (iRc != SUCCESS)
        return ILERROR (iRc, ILXT_ERR_TIF);

    iRc = ILTIFSetPositionAboveTopRecord (tr); // too trivial to ever fail

    gd->m_pRepeatInfo = NULL;
    return SUCCESS;
} //---- CILTranslator::FanBeforeUnload

/*-----
* Function: IL_OutputDebugLog
* Purpose: Write debug message text to log file if
*          Field-Level Logging is turned ON.
* Input:   tr ----- Pointer to ILTR translation structure.
*          pszMsg -- Message text to be logged.
* Return:  void
*-----*/
extern "C"
void IL_DECL IL_OutputDebugLog // Write debug message to log file.
    ( ILTR_PTRANS tr, // Pointer to translation record
      IL_PSTR pszMsg ) // Pointer to message text to be logged
{

```

```

if (ILTR_Flags & ILTR_FIELD_LEVEL_LOGGING)
{
    int    iRc;                // Function return code
    char   szBuffer [ILTB_MAX_MSG + 2]; // Message buffer plus line terminator

    //----- Simply return if no log file exists.
    if (ILTR_hLog == 0 || ILTR_hLog == IL_NULL_HFILE)
        return;

    //----- Copy the message to our internal message buffer.
    IL_STRCPY (szBuffer, pszMsg);

    //----- Add appropriate line terminator to message text string.
    #ifdef ILWIN
        IL_STRCAT (szBuffer, "\r\n");
    #else
        IL_STRCAT (szBuffer, "\n");
    #endif

    //----- Append debug message text to the log file.
    IL_WRITE (ILTR_hLog, szBuffer, IL_STRLEN (szBuffer), iRc);
}

} //----- IL_OutputDebugLog

/*-----
* Name      : MakeLogMessage
* Purpose   : This routine creates a field level debug log message
* Parameters: szMsg -- pointer to message buffer of size ILTB_MAX_MSG
*             pField - pointer to field object
*             szData - character data to go in message buffer
* Returns   : void
*-----*/
void MakeLogMessage                // Make make up field log message
( ILTR_PTRANSL tr,                // Pointer to translation record
  char      *szMsg,                // message buffer, size ILTB_MAX_MSG
  CILField  *pField,               // pointer to field object
  char      *szData )              // field data
{
    if (ILTR_Flags & ILTR_FIELD_LEVEL_LOGGING)
    {
        char      szWork[ILTB_MAX_MSG]; // message work buffer
        int        nNameLen = 0;         // Length of formatted name+label
        int        nFormatChars = 8;     // number of format chars in a msg
        int        nMsgDataLen = 0;      // truncated data length for debug msg

        //----- If field is binary, make up a dummy message for now
        if (pField->m_szType[0] == ILX_TYPE_BINARY)
        {
            IL_SPRINTF (szMsg, "[%s:%s] [Binary]",
                        pField->m_szLabel, pField->m_szName);
            return;
        }

        /*-----
        * Make sure the message will fit in the message buffer.
        * Compute length of name and label including format chars.
        *-----*/
        nNameLen = IL_STRLEN (pField->m_szLabel) +
                    IL_STRLEN (pField->m_szName) + nFormatChars;

        //----- Check if the message will fit in the buffer
        if ((IL_STRLEN (szData) + nNameLen) < ILTB_MAX_MSG)
        {
            IL_SPRINTF (szMsg, "[%s:%s] [%s]", pField->m_szLabel,
                        pField->m_szName, szData);
        }
        else
        {
            /*-----
            * The data is too long to fit in the msg buffer truncate the
            * value to fit in the buffer then move it in.
            *-----*/
            nMsgDataLen = ILTB_MAX_MSG - nNameLen;

```

```

        IL_MEMCPY (szWork, szData, nMsgDataLen);
        szWork [nMsgDataLen] = '\0';

        IL_SPRINTF (szMsg, "[%s:%s] [%s]", pField->m_szLabel,
                    pField->m_szName, szWork);
    }
}
else
    //----- Set message buffer to null string in case some tries to use it
    szMsg[0] = '\0';
} //----- MakeLogMessage

/*-----
* Name:      CallChooseRecords -- static function
* Purpose:   Function to call the Choose Records Dll -- WIN32 only
* Parameters: tr -- Pointer to translation record
* Returns:   IDOK, IDCANCEL, or ILTR_ERR... code
*-----*/
static int CallChooseRecords (ILTR_PTRANSL tr)

//----- When running under WIN32, we invoke the Choose Records dialog
#ifdef WIN32
{
    int      iRc;                // Function return code
    char      szChooseDll [_MAX_PATH]; // Path name to the Choose Record Dll
    PCHOOSE  pChoose;           // Pointer to the ChooseRecords function
    IL_HINST  hChooseDll;       // Handle to the Choose Record Dll

    //----- Get a path name to where ILCHOOSE.DLL is expected to be
    IL_STRCPY (szChooseDll, ILTR_szCurWD);
    if (szChooseDll[IL_STRLEN (szChooseDll) - 1] != IL_FILESEP_CH)
        IL_STRCAT (szChooseDll, IL_FILESEP_STR);
    IL_STRCAT (szChooseDll, "ilchoose.dll");

    //----- Load the Choose Records Dll if available. If not, ignore the call.
    LoadDll (szChooseDll, hChooseDll);
    if (hChooseDll == NULL)
        return IDOK;

    //----- Get the address of the ChooseRecords function
    pChoose = (PCHOOSE) GetProcAddress (hChooseDll, "ChooseRecords");
    if (pChoose == NULL)
        return ILERROR (0, ILTR_ERR_UNKNOWN);

    //----- Now call the Dll ChooseRecords function entry point
    iRc = (*pChoose) (tr);
    return iRc;
} //----- CallChooseRecords

//----- If not running under WIN32, fake it (ALL records are "selected").
#else
{
    return IDOK;
}
#endif

```



```

#if !defined(__ILCILREC)
/*-----
 * Module:  cilrec.h
 * Purpose: Header file for CILRecord class
 * Author:  Dave McConville, Copyright (c) IntelliLink, 1995
 * Notes:   All objects are implemented with pure C++ in order to allow these
 *          class to be platform independent.  No use of MFC or MacApp has been
 *          made.
 *-----*/
#define __ILCILREC          // Indicate header inclusion

//----- Include all the relavent headers
#include "cilfa.h"

/*-----
 * Error codes
 *-----*/

/*-----
 * Maximum values and constant defines
 *-----*/

/*-----
 * type definitions
 *-----*/

/*-----
 * ILRecord class.
 *-----*/
#define CILRecPtr  CILRecord *
class CILRecord
{
    public:

        //----- Class constructor
        CILRecord (CILFldArrayPtr *pFldArray);

        //----- Class destructor
        ~CILRecord ();

        /*-----
         * Caller accessible functions
         *-----*/

        virtual int PutRecord          // Put TIF record to data store
            (ILTR_PTRANSI tr,          // pointer to tr
             CILGLPtr  gd);            // pointer to application data

        virtual int Delete              // Delete an existing record
            (ILTR_PTRANSI tr,          // pointer to tr
             CILGLPtr  gd);            // pointer to application data

        virtual int GetRecord           // Read record from application file
            (ILTR_PTRANSI tr,          // pointer to tr
             CILGLPtr  gd);            // pointer to application data

        virtual int FastSyncDeltaAck    // Acknowledge a FastSync Delta
            (ILTR_PTRANSI tr,          // pointer to tr
             CILGLPtr  gd);            // pointer to application data

        //----- class data members
        CILFldArrayPtr  m_pFldArray;    // pointer to field array
};

#endif // __ILCILREC

```

```

/*-----
* File:      CILREC.CPP
* Purpose:   This module contains the implementation for the member functions
*           of the CILRecord class.
*
* Functions: CILRecord::CILRecord
*           CILRecord::~~CILRecord
*           CILRecord::PutRecord
*           CILRecord::Delete
*           CILRecord::GetRecord
*           CILRecord::FastSyncDeltaAck
*
* Author:    Dave McConville, Copyright (c) IntelliLink Corporation, 1995
*-----*/

#include "cilglobl.h"           // header for CILGlobalData class
#include "cilfield.h"          // header for CILField class
#include "cilfa.h"             // header for CILFieldArray class
#include "cilrec.h"            // header for CILRecord class
#include "cildata.h"           // header for CILData class
#include "ilxtrans.h"
#include "ilxterr.h"           // error codes & error handling protos

//----- local function prototypes
static
int GetRecordUniqueID          // Retrieve record/unique ID(s)
( ILTR_PTRANS tr,              // Pointer to translation record
  CILGLPtr gd );              // Pointer to translator global data

/*-----
* Name       : CILRecord::CILRecord
* Purpose    : Class constructor
* Parameters : pointer to a CILFldArrayPtr to associate with the record
* Returns    : none
*-----*/
CILRecord::CILRecord
( CILFldArrayPtr *pFldArray )
{
    m_pFldArray = *pFldArray;
}

/*-----
* Name       : CILRecord::~~CILRecord
* Purpose    : Class destructor
* Parameters : none
* Returns    : none
*-----*/
CILRecord::~~CILRecord ()
{
}

/*-----
* Name       : CILRecord::PutRecord
* Purpose    : Write record to application file
* Parameters : Pointer to tr, pointer to application data - gd
* Returns    : SUCCESS - all is well
* Notes     :
*-----*/
int CILRecord::PutRecord
( ILTR_PTRANS tr,
  CILGLPtr gd )
{
    int i;                // loop variable
    int iRc;              // Return code
    int nFields;           // number of fields in field array
    ILTR_UPDOPT nAction = UPD_NONE; // Action to take on record
    CILDataStore *pDataStore; // pointer to current data store object
    CILFieldArray *pFldArray; // pointer to field array
    CILField *pField;       // pointer to field object
    long lSize;            // Size of returned data
    IL_PANY pvData;        // Pointer to data value
    char szMsg [ILTB_MAX_MSG]; // Debug message log text

    //----- reference to current field array
    pFldArray = this->m_pFldArray;

```

```

//----- retrieve number of fields.
nFields = pFldArray->m_nFields;

//----- fetch dataStore object from gd
pDataStore = gd->m_pDataStore;

//----- For update, replace, delete or none, get any record ID's from TIF
if ( gd->m_nRecAction == UPD_UPDATE ||
    gd->m_nRecAction == UPD_REPLACE ||
    gd->m_nRecAction == UPD_DELETE ||
    gd->m_nRecAction == UPD_NONE)
{
    //----- Get the record and/or unique ID(s) for this record, if any.
    iRc = GetRecordUniqueID (tr, gd);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_FAIL);
}
else
{
    //----- No record/unique ID's. Reset any current ID values.
    gd->m_ulRecID = (UINT32) -1L;
    gd->m_szRecID[0] = '\0';
    gd->m_szUniqueID[0] = '\0';
}

/*-----
 * Allow data store to do any preparation needed for writing a record
 * before the fields are processed.
 *-----*/
iRc = pDataStore->BeginWriteRecord (tr, gd, gd->m_nRecAction, gd->m_ulRecID);

//----- if the datastore wants to skip this record that's ok
if (iRc >= ILTR_SKIP_WRITE && iRc <= ILTR_SKIP_ALL)
    EXIT_WITH_ERROR (iRc)
else if (iRc)
    //----- any other error is worth logging
    LOG_ERROR_AND_EXIT (iRc, iRc)

//----- Place a new line in the debug log if debug logging.
IL_OutputDebugLog (tr, "");
IL_OutputDebugLog (tr, "Writing Record");

/*-----
 * Need to loop thru the fields reading them from TIF, convert to the
 * applications native format and writing them to the DataStore.
 *-----*/
for (i = 0; i < nFields; i++)
{
    //----- get pointer to current Field object
    pField = pFldArray->At (i);

    //----- is there a field in this slot
    if (pField == NULL)
        continue;

    /*-----
     * Skip any unmapped, non-hidden fields if only mapped fields
     * go in TIF. Otherwise the field will not have been defined
     * in TIF.
     *-----*/
    if ( pDataStore->m_bOnlyMappedInTIF &&
        pField->m_bIsMapped == FALSE &&
        !(pField->m_ulAttribs & ILTB_ATT_HIDDEN_FIELD) )
        continue;

    //----- retrieve TIF field
    iRc = ILTIFGetField (tr, pField->m_szLabel, TIF_AUTO,
                        &lSize, &pvData);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

    //----- save the length of this field's data
    gd->m_uiFieldLen = (unsigned) lSize;

    /*-----
     * At this point in time the data for this field is in the TIF field

```

```

    * buffer. Copy the data to the field buffer in gd.
    *-----*/

//----- will the field fit in the buffer?
if (gd->m_uiFieldLen > gd->m_uiFldBufSize)
{
    //----- is field too big?
    if (gd->m_uiFieldLen > pDataStore->m_uiMaxFldBufSize)
    {
        ILAddFieldError (tr, pField->m_szLabel, ILTR_ERR_TRUNC);
        gd->m_uiFieldLen = pDataStore->m_uiMaxFldBufSize;
    }
    else
    {
        //----- expand the buffer
        gd->m_lpFldBuf = (IL_PSTR) IL_REALLOC (gd->m_uiFieldLen,
                                                gd->m_hFldBuf, gd->m_lpFldBuf);

        //----- was realloc successful?
        if (gd->m_lpFldBuf == NULL)
            EXIT_WITH_ERROR (ILTR_ERR_NOMEM);
    }
}

//----- move the data to the field buffer
IL_MEMCPY (gd->m_lpFldBuf, (IL_PSTR) pvData, gd->m_uiFieldLen);

/*-----
 * Field data has been copied into our local buffer; truncated if
 * necessary to make it fit. Now check to see whether the field
 * definition specifies a max length. If so then truncate the value.
 * NOTE that MaxLength for non-binary fields does NOT include the
 * null terminator, but the length held in "gd->m_uiFldLen" DOES
 * include the null terminator byte.
 *-----*/
if (pField->m_uiMaxLength > 0)
{
    if (pField->m_szType[0] == ILX_TYPE_BINARY)
    {
        //---- check length of binary field; truncate if necessary
        if (gd->m_uiFieldLen > pField->m_uiMaxLength)
        {
            ILAddFieldError (tr, pField->m_szLabel, ILTR_ERR_TRUNC);
            gd->m_uiFieldLen = pField->m_uiMaxLength;
        }
    }

    //---- check length of non-binary field; truncate if necessary
    else if (gd->m_uiFieldLen > pField->m_uiMaxLength+1)
    {
        ILAddFieldError (tr, pField->m_szLabel, ILTR_ERR_TRUNC);
        gd->m_uiFieldLen = pField->m_uiMaxLength+1;
        gd->m_lpFldBuf[pField->m_uiMaxLength] = 0;
    }
}

//----- this field's data is now in field buffer
pField->m_lpData = gd->m_lpFldBuf;

//----- not that field is in IL format
pField->m_nFormat = ILXT_IL_FORMAT;

//----- Make up a log message for the log file.
MakeLogMessage (tr, szMsg, pField, gd->m_lpFldBuf);
IL_OutputDebugLog (tr, szMsg);

//----- convert field and write field to app
iRc = pField->PutToApp (tr, gd);
if (iRc)
    LOG_ERROR_AND_EXIT (iRc, iRc);
}

/*-----
 * Now that all the fields have been handled, write the record to the
 * DataStore.
 *-----*/

```

```

    iRc = pDataStore->EndWriteRecord (tr, gd, gd->m_nRecAction, gd->m_ulRecID);

    //----- if the datastore wants to skip this record that's ok
    if (iRc >= ILTR_SKIP_WRITE && iRc <= ILTR_SKIP_ALL)
        EXIT_WITH_ERROR (iRc)
    else if (iRc)
        //----- any other error is worth logging
        LOG_ERROR_AND_EXIT (iRc, iRc);

    //----- all is well
    iRc = SUCCESS;

Exit:
    //----- All done
    return (iRc);
}

/*-----
* Name      : CILRecord::Delete
* Purpose   : This routine is called to perform a delete operation.
* Parameters : Pointer to tr, pointer to application data - gd
* Returns   : SUCCESS - all is well
*           : ILXT_ERR_FAIL - couldn't get Unique ID from TIF
*           : or error returned by the DataStore DeleteRecord function
* Notes     : This function fetches the record id from the current TIF
*           : record then calls the DataStore object to delete the rec.
*-----*/
int CILRecord::Delete
    (ILTR_PTRANSI tr,
     CILGLPtr     gd)
{
    int          iRc;                // Return code
    CILDataStore * pDataStore;       // pointer to data store

    //----- Get the record and/or unique ID(s) for this record, if any.
    iRc = GetRecordUniqueID (tr, gd);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_FAIL);

    //----- Fetch the DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- Delete the record from the DataStore
    iRc = pDataStore->DeleteRecord (tr, gd);

Exit:
    //----- All done
    return (iRc);
}

/*-----
* Name      : CILRecord::GetRecord
* Purpose   : Read record from application file
* Parameters : Pointer to tr, pointer to application data - gd
* Returns   : SUCCESS - all is well
*           : others as returned from CILField::GetFromApp
* Notes     :
*-----*/
int CILRecord::GetRecord
    (ILTR_PTRANSI tr,
     CILGLPtr     gd)
{
    int          iRc;                // Return code
    int          i;                  // loop variable
    int          nFields;            // number of fields in field array
    CILField     * pField;           // pointer to field object
    char         szMsg [ILTB_MAX_MSG]; // Debug message log text

    //----- initialization for new record
    gd->m_lStartDate = 0L;
    gd->m_bSkipToDo = FALSE;

    //----- get number of fields in field array.

```

```

nFields = this->m_pFldArray->m_nFields;

//----- Place a new line in the debug log if debug logging.
IL_OutputDebugLog (tr, "");
IL_OutputDebugLog (tr, "Reading Record");

//----- loop thru the fields and get the data
for (i = 0; i < nFields; ++i)
{
    //----- fetch the next field in the field array
    pField = this->m_pFldArray->At (i);

    //----- is there a field in this slot?
    if (pField == NULL)
        continue;

    //----- empty field buffer
    gd->m_lpFldBuf[0] = '\0';
    gd->m_uiFieldLen = 0;

    //----- tell the field to give up its value
    iRc = pField->GetFromApp (tr, gd);
    if (iRc)
    {
        //--- log abnormal error for this field number
        ILERROR (i, iRc);
        break;
    }

    //----- Make up a log message for the log file.
    MakeLogMessage (tr, szMsg, pField, gd->m_lpFldBuf);

    //----- debug message is safely in the buffer
    IL_OutputDebugLog (tr, szMsg);
}

//----- perform range checking for To Do items if range has been specified
if (gd->m_bToDoRangeCheck)
{
    //----- does this to do repeat?
    if (gd->m_bRepeatingAppt)
    {
        //----- skip if start is higher than end of range
        if (gd->m_pRepeatInfo->startDate > ILTR_nHiDate)
            gd->m_bSkipToDo = TRUE;
        //----- skip if there is a stop date & its lower than begin range
        else if ((gd->m_pRepeatInfo->stopDate) &&
            (gd->m_pRepeatInfo->stopDate < ILTR_nLoDate))
            gd->m_bSkipToDo = TRUE;
    }
    else if (gd->m_lStartDate) // its a single to do with a start date
    {
        //----- check if the start date is in specified range
        if ((gd->m_lStartDate < ILTR_nLoDate) ||
            (gd->m_lStartDate > ILTR_nHiDate))
        {
            gd->m_bSkipToDo = TRUE;
        }
    }
}

//----- If we encountered a done or out of range ToDo, skip this record.
if (gd->m_bSkipToDo)
{
    ILSetAction (tr, ILTR_ACT_RANGE);
    return ILTR_SKIP_WRITE;
}

return iRc;
}

/*-----
* Name      : CILRecord::FastSyncDeltaAck
*
* Purpose   : To acknowledge a single record's FastSync Delta.

```

```

*
* Explanation: Some but not all translators have an advanced capability
*              called "FastSync". When a FastSync-capable translator
*              is operating in FastSync mode, all the records that it
*              loads into TIF are DELTA (i.e. change) records, where
*              the DELTA code (value of the _Delta field) is one of
*              the values A for Add, C for Change, or D for Delete.
*
*              Sometimes FastSync translators operate in SlowSync mode, in
*              which case all the un-deleted records that are known to
*              the translator must be assigned status=NEW.
*
*              Even after loading a delta into TIF, the translator
*              remains responsible for remembering and potentially
*              re-supplying the same delta until the
*              "FastSyncDeltaAck" function is called FOR THAT RECORD, during
*              the unload-from-TIF phase. This function should "clear
*              status bits" or do whatever it takes to make the
*              delta be permanently dead and gone.
*
*              FastSyncDeltaAck is called (during unload) to ACK inputs
*              from FastSync-capable translators, regardless of whether the
*              LOAD phase is done in FastSync or SlowSync mode. In either
*              case the possibility exists that some inputs may remain
*              unconsumed, and will need to be re-supplied the next time
*              synchronization is performed.
*
*              Note that Unique IDs are required for Fast Sync.
*
* Parameters : Pointer to tr, pointer to application data - gd
* Returns    : SUCCESS - all is well
*              others as returned from underlying functions
* Notes      :
*-----*/
int CILRecord::FastSyncDeltaAck
    (ILTR_PTRANSL tr,
     CILGIPtr     gd)
{
    int          iRc;          // Return code

    //----- Get the unique ID(s) for this record, if any.
    iRc = GetRecordUniqueID (tr, gd);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_FAIL);

    //----- Pass the Delta ACK on to the DataStore
    iRc = gd->m_pDataStore->AckDelta (tr, gd);

Exit:
    //----- All done
    return (iRc);
}

/*-----
* Name:      GetRecordUniqueID -- internal function
* Purpose:   Retrieve record and unique ID's (if any) from the TIF database
*            in preparation for a delete, update, or record record operation.
* Parameters Pointer to tr, pointer to application data - gd
* Returns    SUCCESS - all is well
*            ILXT_ERR_TIF - some problem reading TIF field data
*-----*/
int GetRecordUniqueID
    ( ILTR_PTRANSL tr,          // Retrieve record/unique id(s)
      CILGIPtr     gd )        // Pointer to translation record
    {                          // Pointer to translator global data
    int          iRc = SUCCESS; // Return code
    long         lSize;         // Size of returned data
    IL_PANY      pvData;        // Pointer to data value
    IL_PSTR      pszEnd;        // Pointer to any text after record ID

    //----- Reset any current record and unique ID values.
    gd->m_ulRecID = (UINT32) -1L;
    gd->m_szRecID[0] = '\0';
    gd->m_szUniqueID[0] = '\0';
}

```

```

//----- If we have record ID's get the record ID from original TIF record.
if (gd->m_bHasRecordID)
{
    iRc = ILTIFGetField (tr, ILTR_FLD_RECORD_ID, TIF_ORIGINAL, &lSize, &pvData);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

    //----- Convert record ID to numeric record ID and save it in gd.
    gd->m_ulRecID = strtoul ((IL_PSTR) pvData, &pszEnd, 10);

    //----- Copy the record ID string into global data.
    if (lSize <= ILXT_MAX_RECID_STRING)
        IL_STRCPY (gd->m_szRecID, (char *) pvData);
    else
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_LIMIT);
}

//----- If we have unique ID's get the unique ID from original TIF record.
if (gd->m_bHasUniqueID)
{
    iRc = ILTIFGetField (tr, ILTR_FLD_UNIQUE_ID, TIF_ORIGINAL, &lSize, &pvData);
    if (iRc != SUCCESS)
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_TIF);

    //----- Copy the UNIQUE record ID string into global data.
    if (lSize <= ILXT_MAX_RECID_STRING)
        IL_STRCPY (gd->m_szUniqueID, (char *) pvData);
    else
        LOG_ERROR_AND_EXIT (iRc, ILXT_ERR_LIMIT);
}

Exit:
//----- All done
return (iRc);
}

```



```

#if !defined(__ILCILFIELD)
/*-----
 * Module:  cilfield.h
 * Purpose: Header file for field class
 * Author:  Dave McConville, Copyright (c) IntelliLink, 1995
 * Notes:
 * Libraries: The following libraries are needed to use ILXL:
 *           <None>
 *-----*/
#define __ILCILFIELD          // Indicate header inclusion

//----- Include all the relevant headers
#include "cilglobl.h"
#include "iltbl.h"
#include "iltr.h"

/*-----
 * ILField class.
 *-----*/
#define CILFieldPtr  CILField *
class CILField
{
public:

    //----- Class constructor
    CILField ();

    //----- Class destructor
    ~CILField ();

    /*-----
     * Caller accessible functions
     *-----*/

    int Type
        (char          * szBuf,          // return type of field
         unsigned int uiBufSize);         // string buffer
                                         // size of buffer

    int SetType
        (char * szType);                 // set the field type
                                         // ptr to type string

    BOOLEAN IsType
        (char cType);                   // set the field type
                                         // type as char

    virtual int GetFromApp
        (ILTR_PTRANSI tr,                // move field from app to IL
         CILGLPtr      gd);              // pointer to tr
                                         // pointer to global data object

    virtual int PutToIL
        ( ILTR_PTRANSI tr,                // wrapper for ILFldPut
         IL_PANY pVal,
         UINT32 len );

    virtual int ProcessGet
        (ILTR_PTRANSI tr,                // perform action on field for get
         CILGLPtr      gd);              // pointer to tr
                                         // pointer to application data

    virtual int PutToApp
        (ILTR_PTRANSI tr,                // Move field from IL to app
         CILGLPtr      gd);              // pointer to tr
                                         // pointer to global data object

    virtual int ProcessPut
        (ILTR_PTRANSI tr,                // perform action on field for get
         CILGLPtr      gd);              // pointer to tr
                                         // pointer to global data object

    //----- field object attributes

    int          m_nILTRIndex;           // index in ILTR Field List
    int          m_nCILFAIndex;          // index in Field Array
    char         m_szLabel               // internal field name
        [ILTR_MAX_FLDNAME + 1];         // ..
    char         m_szName               // external field name
        [ILTR_MAX_FLDNAME + 1];         // ..
}

```

```
char      m_szType [2];           // data type of field
char      m_szDefaultValue       // Default value (if any)
        {ILTR_MAX_TYPEDESC};    // ..
unsigned int m_uiMaxLength;       // max length for data
                                   // 0 = variable length
ILX_ATTR  m_ulAttribs;           // Field attributes
int        m_nFieldAction;       // Action to be taken on field
BOOLEAN    m_bIsMapped;         // whether field is mapped
IL_PSTR    m_lpData;            // pointer to field data
int         m_nFormat;           // format data is stored in
int         m_nOrigin;           // field origin
long        m_lRefCon;           // each xlator is free to use
                                   // this field for whatever !!
};

#endif // __ILCILFIELD
```

```

/*-----
* File:      CILFIELD.CPP
* Purpose:   Implementation of member functions for CILFeild class
* Functions: CILField::CILField
*           CILField::~CILField
*           CILField::Type
*           CILField::SetType
*           CILField::GetFromApp
*           CILField::PutToIL
*           CILField::ProcessGet
*           CILField::PutToApp
*           CILField::ProcessPut
* Author:    Dave McConville, Copyright (c) IntelliLink, 1995
*-----*/

//----- Include all the relavent headers
#include "cilglobl.h"      // header for CILGlobalData class
#include "cilfield.h"      // header for CILField class
#include "cildata.h"       // header for CILDataStore class
#include "ilctr.h"         // general ILTR header file
#include "ilxtrans.h"
#include "ilxterr.h"       // error codes & error handling protos

extern "C"
{
#include "ilmacro.h"
}

/*-----
* Name       : CILField::CILField
* Purpose    : Class constructor.  Initializes data members.
* Parameters : none
* Returns    : none
*-----*/
CILField::CILField ()
{
    m_nILTRIndex = -1;           // Field's index in ILTR Field List
    m_nCILFAIndex = -1;         // Field's index in FieldArray
    m_szLabel[0] = ILXT_CHAR_NULL; // internal field name
    m_szName[0] = ILXT_CHAR_NULL; // external field name
    m_szType[0] = 'A';          // data type of field
    m_szType[1] = ILXT_CHAR_NULL; // ..
    m_szDefaultValue[0] = ILXT_CHAR_NULL; // Default field value (if any)
    m_uiMaxLength = 0;          // max length for data; 0 = variable
    m_ulAttribs = 0;            // Field attributes
    m_nFieldAction = ILXT_ACT_NORMALFIELD; // Action to be taken on this field
    m_bIsMapped = FALSE;        // whether field is mapped
    m_lpData = NULL;            // pointer to field data
    m_nFormat = ILXT_UNKNOWN_FORMAT; // true if data is stored in IL format
    m_nOrigin = ILXT_APP_FIELD; // field origin (app or "special")
    m_lRefCon = -1;             // xlator-defined usage
}

/*-----
* Name       : CILField::~CILField
* Purpose    : Class destructor
* Parameters : none
* Returns    : none
*-----*/
CILField::~CILField ()
{
}

/*-----
* Name       : CILField::Type
* Purpose    : return type of field
* Parameters : szBuf - string buffer to copy type into
*           nBufSize - size of buffer passed
* Returns    : ILXT_OK - all is well
*           ILXT_ERR_TRUNC - incoming type string was truncated
*-----*/
int CILField::Type
(char * szBuf,           // string buffer
 unsigned int uiBufSize) // size of buffer

```

```

(
    if (IL_STRLEN (this->m_szType) > uiBufSize)
    {
        IL_MEMCPY (szBuf, this->m_szType, uiBufSize);
        return ILXT_ERR_TRUNC;
    }
    else
    {
        IL_STRCPY (szBuf, this->m_szType);
        return ILXT_OK;
    }
}

/*-----
* Name      : CILField::SetFieldType
* Purpose   : sets the field type
* Parameters : szType - pointer to type string
* Returns   : ILXT_OK
*           : ILXT_ERR_BAD_FLD_TYPE
*-----*/
int CILField::SetType
    (char * szType)          // ptr to type string
{
    if (IL_STRLEN (szType) > ILTR_MAX_TYPEDESC)
    {
        this->m_szType[0] = ILXT_CHAR_NULL;
        return ILXT_ERR_BAD_FLD_TYPE;
    }
    else
    {
        IL_STRCPY (this->m_szType, szType);
        return ILXT_OK;
    }
}

/*-----
* Name      : CILField::IsType
* Purpose   : check if field is of the given type
* Parameters : cType - type to check for
* Returns   : True | False
*-----*/
BOOLEAN CILField::IsType
    (char cType)            // field type as char
{
    if (this->m_szType[0] == cType)
        return TRUE;
    else
        return FALSE;
}

/*-----
* Name      : CILField::GetFromApp
* Purpose   : Retrieve field from application and put field to ILTR
* Parameters : Pointer to ILTR global, pointer to global data object
* Returns   : SUCCESS
*           : error code from ILFldPut or ILTIFPutField
*-----*/
int CILField::GetFromApp
    (ILTR_PTRANSL tr,
     CILGIPtr gd)
{
    CILDataStore * pDataStore;          // pointer to DataStore object
    int iRc;                            // return code

    //----- fetch the DataStore object from gd
    pDataStore = gd->m_pDataStore;

    /*-----
    * If this field is not an IntelliLink Special field, then
    * read the field from the DataStore, else just allow the
    * special processing to take place in the ProcessGet
    * member function.
    *-----*/
    if (m_nOrigin == ILXT_APP_FIELD)
    {
        //----- fetch the next field from the current record

```

```

    iRc = pDataStore->ReadField (tr, gd, this);
    if (iRc)
        EXIT_WITH_ERROR (iRc);
}

//----- data is in field buffer and in application "native" format.
m_lpData = gd->m_lpFldBuf;
m_nFormat = ILXT_NATIVE_FORMAT;

//----- process the field based on its action code
iRc = this->ProcessGet (tr, gd);
if (iRc)
    EXIT_WITH_ERROR (iRc);

//----- If we have valid field data, write it to intermediate file now
if (gd->m_nFieldAction == ILXT_PUTFIELD)
{
    /*-----
    * Only put out this field if we have a non-null buffer pointer.
    * NOTE: we do put out zero-length fields here; required for FILTERS
    * to work right. It is probably impossible, or at least abnormal,
    * to have a null buffer pointer here.
    *-----*/
    if (m_lpData)
    {
        //----- is this a real export or export before import
        if (gd->m_nPhase == ILXT_EXPORT)
        {
            //----- it's an export : write to ILIF or TIF or Transfer File
            iRc = this->PutToIL (tr, m_lpData, (UINT32) gd->m_uiFieldLen);

            //----- check for valid return codes - their ok
            if ((iRc == ILIF_ERR_NOFIELD) || (iRc == ILTR_ERR_NOFLD))
                iRc = SUCCESS;

            //----- If truncated, flag as field error and clear
            if (iRc == ILTR_ERR_TRUNC)
            {
                ILAddFieldError (tr, m_szLabel, ILTR_ERR_TRUNC);
                iRc = SUCCESS;
            }
        }
        else if (gd->m_nPhase == ILXT_EXPORT_BEFORE_IMPORT)
        {
            /*-----
            * This is an export before import. Skip this field if only mapped
            * fields are supposed to go into TIF and this field is not mapped
            * and this is NOT a HIDDEN field. All HIDDEN fields go to TIF.
            *-----*/
            if (pDataStore->m_bOnlyMappedInTIF && (!m_bIsMapped) &&
                !(m_ulAttribs & ILTB_ATT_HIDDEN_FIELD) )
                ; // do nothing - skip the write
            else
            {
                //----- write the field to TIF
                iRc = ILTIFPutField (tr, m_szLabel, m_lpData, gd->m_uiFieldLen);
                if (iRc)
                    LOG_ERROR_AND_EXIT (iRc, iRc);
            }
        }
        else
        {
            //----- Logic error if not exporting or exporting before import
            LOG_ERROR_AND_EXIT (0, ILXT_ERR_LOGIC);
        }
    }
}
else // don't write field to ILIF
{
    //----- reset Field Action to default
    gd->m_nFieldAction = ILXT_PUTFIELD;
}

Exit:
//----- all done with this field
return iRc;

```

```

} //----- CILField::GetFromApp

/*-----
* Name      : CILField::PutToIL
*-----*/
int CILField::PutToIL
    ( ILTR_PTRANSI tr,
      IL_PANY pVal,
      UINT32 len )
{
    int iRc;

    iRc = ILFldPut (tr, m_szLabel, (IL_PSTR) pVal, (unsigned int) len);
    return iRc;
} //----- CILField::PutToIL

/*-----
* Name      : CILField::ProcessGet
* Purpose   : Process the field based on it FieldAction code
* Parameters : Pointer to ILTR global, pointer to global data object
* Returns   : SUCCESS - all is well
*            ILXT_ERR_UNK_VAL - unknown field action code
*            return codes from DataStore member functions
*-----*/
int CILField::ProcessGet
    ( ILTR_PTRANSI tr,          // pointer to tr
      CILGlPtr      gd)        // pointer to application data
{
    int          iRc = SUCCESS;    // return code
    CILDataStore *pDataStore;      // pointer to DataStore object

    //----- fetch DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- Set default action to "put this field" to intermediate file
    gd->m_nFieldAction = ILXT_PUTFIELD;

    //----- Assume data is in the field buffer unless told otherwise
    m_lpData = gd->m_lpFldBuf;

    /*-----
    * If this is an application field (i.e. not hidden or special)
    * and there is no data, then don't bother processing.
    *-----*/
    if ((m_nOrigin == ILXT_APP_FIELD) && (gd->m_uiFieldLen == 0))
    {
        return SUCCESS;
    }

    //----- Process the field in accordance with its "action" code
    switch (m_nFieldAction)
    {
        case ILXT_ACT_STARTDATE:          // Field is a "start date"
        case ILXT_ACT_ENDDATE:            // Field is an "end date"
        case ILXT_ACT_ALARMDATE:          // Field is an "alarm date"
            //----- allow datastore to convert dates
            m_lpData = pDataStore->DateToIL (m_lpData, gd->m_lpConvBuf);

            //----- recalculate the field length
            gd->m_uiFieldLen = IL_STRLEN (m_lpData);

            //----- is this the start date?
            if (m_nFieldAction == ILXT_ACT_STARTDATE)
            {
                /*-----
                * If there is a value for this field then convert it
                * from Alpha Date format to Date Code (long) format
                * and save it away for range checking.
                *-----*/
                if (m_lpData[0] != '\0')
                {
                    //----- convert it and save it
                    gd->m_lStartDate = IL_AlphaToCodeDate ((IL_PSTR) m_lpData);
                }
            }
        }
    }

```

```

        else
            //----- specify no start date
            gd->m_lStartDate = 0;
    }
    break;

case ILXT_ACT_STARTTIME:                // Field is a "start time"
case ILXT_ACT_ENDTIME:                  // Field is an "end time"
case ILXT_ACT_ALARMTIME:                // Field is an "alarm time"
    //----- allow datastore to convert times
    m_lpData = pDataStore->TimeToIL (m_lpData, gd->m_lpConvBuf);

    //----- recalculate the field length
    gd->m_uiFieldLen = IL_STRLEN (m_lpData);
    break;

case ILXT_ACT_ALARMFLAG:                // Field is an "alarm flag"
    //----- allow datastore to convert Boolean flags
    m_lpData = pDataStore->BoolToIL(m_lpData, gd->m_lpConvBuf);

    //----- recalculate the field length
    gd->m_uiFieldLen = IL_STRLEN (m_lpData);
    break;

case ILXT_ACT_PHONE:                    // Field is a "phone field"
    //----- allow datastore to convert phones
    m_lpData = pDataStore->PhoneToIL (m_lpData, gd->m_lpConvBuf);

    //----- recalculate the field length
    gd->m_uiFieldLen = IL_STRLEN (m_lpData);
    break;

case ILXT_ACT_PRIORITY:                 // Field is a "priority"
    //----- allow datastore to convert priorities
    m_lpData = pDataStore->PriorToIL (m_lpData, gd->m_lpConvBuf);

    //----- recalculate the field length
    gd->m_uiFieldLen = IL_STRLEN (m_lpData);
    break;

case ILXT_ACT_TODODONEFLD:              // Field is "To Do Completed"
    //----- allow datastore to convert flag
    m_lpData = pDataStore->BoolToIL (m_lpData, gd->m_lpConvBuf);

    //----- recalculate the field length
    gd->m_uiFieldLen = IL_STRLEN (m_lpData);

    //----- Make sure we have a valid pointer before using it
    if (m_lpData == NULL)
        iRC = ILERROR (0, ILXT_ERR_LOGIC);

    //----- should Done To Do check be performed?
    if (gd->m_bToDoDoneCheck)
    {
        //----- Set skip flag if Todo done & only exporting not done Todo's
        if (ILTR_nSchOpt == ILTB_TODO_NOTDONE && m_lpData[0] == '1')
            gd->m_bSkipToDo = TRUE;
    }
    break;

case ILXT_ACT_KEYFIELD:                 // Field is a "key field"
case ILXT_ACT_NORMALFIELD:              // Field is a "normal field"
    //----- Allow datastore to convert data according to its type
    if (this->IsType (ILX_TYPE_BOOL))
    {
        //----- convert BOOLEAN to IntelliLink format
        m_lpData = pDataStore->BoolToIL (m_lpData, gd->m_lpConvBuf);
    }
    else if (this->IsType (ILX_TYPE_DATE))
    {
        //----- convert Date to IntelliLink format
        m_lpData = pDataStore->DateToIL (m_lpData, gd->m_lpConvBuf);
    }
    else if (this->IsType (ILX_TYPE_TIME))
    {
        //----- convert time to IntelliLink format

```

```

        m_lpData = pDataStore->TimeToIL (m_lpData, gd->m_lpConvBuf);
    }
    else if (this->m_ulAttribs & ILTB_ATT_VIEW)
        ILSetRecName (tr, m_lpData);

    //----- recalculate the field length
    gd->m_uiFieldLen = IL_STRLEN (m_lpData);
    break;

case ILXT_ACT_APPDATA:
    /*-----
     * This special field allows applications to pass arbitrary data
     * to the intermediate file. Allow the DataStore to use this field
     * by processing this action and calling the DataStore's Get_appdata
     * member function. Data (if any) is placed in gd->m_lpFldBuf.
     *-----*/
    iRc = pDataStore->Get_appdata (tr, gd, this);
    m_lpData = gd->m_lpFldBuf;
    break;

case ILXT_ACT_REPEATBASIC:
    /*-----
     * This special field is used to extract repeat information for the
     * current record from the DataStore. The DataStore's GetRepeat
     * member function is responsible for setting the m_bRepeatingAppt
     * flag to specify whether or not this record repeats. It is also
     * the responsibility of that function to fill in the ILTR_REPEAT
     * structure pointed to by gd if it is in fact a repeating item.
     *-----*/
    iRc = pDataStore->GetRepeat (tr, gd, this);
    gd->m_nFieldAction = ILXT_IGNOREFIELD;
    break;

case ILXT_ACT_REPEATXDATES:
    /*-----
     * This special field is treated together with the "Repeat Basic"
     * field under ILXT_ACT_REPEATBASIC, and can be ignored here.
     *-----*/
    gd->m_nFieldAction = ILXT_IGNOREFIELD;
    break;

case ILXT_ACT_RECORDID:
    /*-----
     * This special field is used to allow the data store to assign an
     * id to this record.
     *-----*/
    iRc = pDataStore->GetRecID (tr, gd, this);
    m_lpData = gd->m_lpFldBuf;
    break;

case ILXT_ACT_UNIQUEID:
    /*-----
     * This special field is used to allow the data store to assign
     * a UNIQUE id to this record.
     *-----*/
    iRc = pDataStore->GetUniqueID (tr, gd, this);
    m_lpData = gd->m_lpFldBuf;
    break;

case ILXT_ACT_DELTA:
    /*-----
     * This special field is used, for "Fast Sync" translators only,
     * to allow such translators to pinpoint Adds, Changes, and Deletes
     * that have occurred since the last time synchronization was done.
     *-----*/
    iRc = pDataStore->GetDelta (tr, gd, this);
    m_lpData = gd->m_lpFldBuf;
    break;

case ILXT_ACT_SUBTYPE:
    /*-----
     * This special field carries the Section SubType Tag (aka Origin Tag)
     * from Source to Target, but translators should not touch this field
     * directly. Special harness code moves Tags back and forth between
     * the "_subType" field and user-visible fields which have
     * field attribute ILTB_ATT_TAGGED.
    */

```



```

        *-----*/
        gd->m_nFieldAction = ILXT_IGNOREFIELD;
        break;

    default:
        //----- Is this a special DataStore action
        if (m_nFieldAction >= ILXT_ACT_USER_BASE)
            //----- pass it along to the DataStore
            iRc = pDataStore->ProcessGetAction (tr, gd, this);
        else
            //----- unknown field action code
            iRc = ILERROR (m_nFieldAction, ILXT_ERR_UNK_VAL);

        //----- if there was an error then ignore this field
        if (iRc)
            gd->m_nFieldAction = ILXT_IGNOREFIELD;
    }

    //----- Make sure we have a valid data pointer if "put field" is set.
    if (m_lpData == NULL && gd->m_nFieldAction == ILXT_PUTFIELD)
        iRc = ILERROR (0, ILXT_ERR_LOGIC);

    //----- we are done
    return iRc;
} //---- CILField::ProcessGet

/*-----
* Name      : CILField::PutToApp
* Purpose   : Move field from IL to app
* Parameters : Pointer to ILTR global, pointer to global data object
* Returns   : SUCCESS or error code
*-----*/
int CILField::PutToApp
    (ILTR_PTRANSL tr,          // pointer to tr
     CILGLPtr      gd)        // pointer to application data
{
    CILDataStore * pDataStore;    // pointer to DataStore object
    int          iRc;             // return code

    //----- fetch DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- process each field based on its field type and attributes
    iRc = this->ProcessPut (tr, gd);
    if (iRc)
        EXIT_WITH_ERROR (iRc);

    //----- data is currently in application "native" format
    m_nFormat = ILXT_NATIVE_FORMAT;

    /*-----
    * If this field is not an IntelliLink Special field, then write
    * the field to the DataStore, else the special processing that
    * took place in ProcessPut member function should have handled it.
    *-----*/
    if (m_nOrigin == ILXT_APP_FIELD)
        //----- Write the field to the record
        iRc = pDataStore->WriteField (tr, gd, this);

Exit:
    //---- all done
    return iRc;
} //---- CILField::PutToApp

/*-----
* Name      : CILField::ProcessPut
* Purpose   :
* Parameters : Pointer to ILTR global, pointer to global data object
* Returns   : SUCCESS or error code
*-----*/
int CILField::ProcessPut
    (ILTR_PTRANSL tr,          // pointer to tr

```

```

        CILGlPtr    gd)           // pointer to application data
(
    int             iRc = SUCCESS;    // return code
    int             nExSize;          // size of repeat exlusion list (bytes)
    long            lSize;            // Return value from TIF
    IL_PANY         _pvData;          // Return value from TIF
    CILDataStore    *pDataStore;      // pointer to DataStore object
    IL_PSTR         pOriginalData;    // original pointer to field data

    //----- fetch DataStore object from gd
    pDataStore = gd->m_pDataStore;

    //----- remember original pointer to data
    pOriginalData = m_lpData;

    //----- Process the field in accordance with its "action" code
    switch (m_nFieldAction)
    {
        case ILXT_ACT_STARTDATE:           // Field is a "start date"
        case ILXT_ACT_ENDDATE:             // Field is an "end date"
        case ILXT_ACT_ALARMDATE:           // Field is an "alarm date"
            //----- allow DataStore to convert to native date format
            m_lpData = pDataStore->DateFromIL (m_lpData, gd->m_lpConvBuf);

            //----- was a conversion successful?
            if (m_lpData == NULL)
                iRc = ILERROR (0, ILXT_ERR_CONV);

            break;

        case ILXT_ACT_STARTTIME:           // Field is a "start time"
        case ILXT_ACT_ENDTIME:             // Field is an "end time"
        case ILXT_ACT_ALARMTIME:           // Field is an "alarm time"
            //----- allow DataStore to convert to native time format
            m_lpData = pDataStore->TimeFromIL (m_lpData, gd->m_lpConvBuf);

            //----- was a conversion successful?
            if (m_lpData == NULL)
                iRc = ILERROR (0, ILXT_ERR_CONV);

            break;

        case ILXT_ACT_ALARMFLAG:           // Field is an "alarm flag"
            //----- allow DataStore to convert boolean
            m_lpData = pDataStore->BoolFromIL(m_lpData, gd->m_lpConvBuf);

            //----- was a conversion successful?
            if (m_lpData == NULL)
                iRc = ILERROR (0, ILXT_ERR_CONV);

            break;

        case ILXT_ACT_PHONE:               // Field is a "phone field"
            //----- allow DataStore to convert phone
            m_lpData = pDataStore->PhoneFromIL (m_lpData, gd->m_lpConvBuf);

            //----- was a conversion successful?
            if (m_lpData == NULL)
                iRc = ILERROR (0, ILXT_ERR_CONV);

            break;

        case ILXT_ACT_PRIORITY:            // Field is a "priority"
            //----- allow DataStore to convert priorities
            m_lpData = pDataStore->PriorFromIL (m_lpData, gd->m_lpConvBuf);

            //----- was a conversion successful?
            if (m_lpData == NULL)
                iRc = ILERROR (0, ILXT_ERR_CONV);

            break;

        case ILXT_ACT_TODOONEFLD:         // Field is "To Do Completed"
            //----- allow DataStore to convert boolean
            m_lpData = pDataStore->BoolFromIL (m_lpData, gd->m_lpConvBuf);
    }

```

```

//----- was a conversion successful?
if (m_lpData == NULL)
    iRc = ILERROR (0, ILXT_ERR_CONV);

break;

case ILXT_ACT_KEYFIELD: // Field is a "key field"
case ILXT_ACT_NORMALFIELD: // Field is a "normal field"
    if (! m_lpData)
        break;
    if (this->IsType (ILX_TYPE_BOOL))
        //----- allow DataStore to convert boolean
        m_lpData = pDataStore->BoolFromIL (m_lpData, gd->m_lpConvBuf);
    else if (this->IsType (ILX_TYPE_DATE))
        //----- allow DataStore to convert date
        m_lpData = pDataStore->DateFromIL (m_lpData, gd->m_lpConvBuf);
    else if (this->IsType (ILX_TYPE_TIME))
        //----- allow DataStore to convert time
        m_lpData = pDataStore->TimeFromIL (m_lpData, gd->m_lpConvBuf);
    else if (this->m_ulAttribs & ILTB_ATT_VIEW)
        ILSetRecName (tr, m_lpData);

//----- was a conversion successful?
if (m_lpData == NULL)
    iRc = ILERROR (0, ILXT_ERR_CONV);

break;

case ILXT_ACT_APPDATA:
/*-----
 * This special field allows applications to pass arbitrary data
 * to the intermediate file. Allow the DataStore to use this field
 * by processing this action and calling the DataStore's Put_appdata
 * member function.
 *-----*/
    iRc = pDataStore->Put_appdata (tr, gd, this);
    break;

case ILXT_ACT_REPEATBASIC:
/*-----
 * This special field is used to pass repeat information for the
 * current record to the DataStore. The DataStore's PutRepeat
 * member function is responsible for extracting the data from
 * the ILTR_REPEAT structure pointed to by gd if this record is
 * in fact a repeating item.
 *-----*/

//----- check if there is a repeat structure for this record
if (gd->m_uiFieldLen == 0)
    break;

//----- copy repeat struct from field buffer
IL_MEMCPY (gd->m_pRepeatInfo, gd->m_lpFldBuf, sizeof (ILTR_REPEAT));

//----- if there are exclusions then create exclusion list
if (gd->m_pRepeatInfo->numExDates)
{
    //----- determine size of exclusion list
    nExSize = gd->m_pRepeatInfo->numExDates * sizeof (long);

    //----- allocate space for exclusion list
    gd->m_pRepeatInfo->exDates = (ILTR_PDATES) IL_ALLOC (nExSize,
                                                         gd->m_pRepeatInfo->hExDates);
    if (gd->m_pRepeatInfo->exDates == NULL)
        return ILTR_ERR_NOMEM;

    //----- Get the exclusion list from TIF
    iRc = ILTIFGetField (tr, ILTR_REP_XDATE, TIF_AUTO,
                        &lSize, &pvData);

    //----- These had better agree!
    if (lSize != (long)nExSize)
        return ILERROR (nExSize, ILXT_ERR_USAGE);

    //----- copy the exclusion list from field buffer
    IL_MEMCPY (gd->m_pRepeatInfo->exDates, (IL_PSTR)pvData, nExSize);
}

```

```

    }

    //----- let the datastore handle the repeat stuff
    iRc = pDataStore->PutRepeat (tr, gd, this);
    break;

case ILXT_ACT_REPEATXDATES:
    /*-----
     * This special field is treated together with the "Repeat Basic"
     * field under ILXT_ACT_REPEATBASIC, and can be ignored here.
     *-----*/
    break;

case ILXT_ACT_RECORDID:
    /*-----
     * This special field is used to store the record id for a particular
     * record. Pass this id onto the datastore for current record.
     *-----*/
    iRc = pDataStore->PutRecID (tr, gd, this);
    break;

case ILXT_ACT_UNIQUEID:
    /*-----
     * This special field is used to set the UNIQUE record id for a
     * record. Pass this id onto the datastore for current record.
     *-----*/
    iRc = pDataStore->PutUniqueID (tr, gd, this);
    break;

case ILXT_ACT_DELTA:
    /*-----
     * This special field is used, for "Fast Sync" translators only,
     * to allow such translators to pinpoint Adds, Changes, and Deletes
     * that have occurred since the last time synchronization was done.
     * It is meaningful for EXPORT only, so we do nothing with it here.
     *-----*/
    iRc = SUCCESS;
    break;

case ILXT_ACT_SUBTYPE:
    /*-----
     * This special field carries the Section SubType Tag (aka Origin Tag)
     * from Source to Target, but translators should not touch this field
     * directly. Special harness code moves Tags back and forth between
     * the "_subType" field and user-visible fields which have
     * field attribute ILTB_ATT_TAGGED.
     *-----*/
    iRc = SUCCESS;
    break;

default:
    //----- Is this a special DataStore action
    if (m_nFieldAction >= ILXT_ACT_USER_BASE)
        //----- pass it along to the DataStore
        iRc = pDataStore->ProcessPutAction (tr, gd, this);
    else
        //----- unknown field action code
        iRc = ILERROR (0, ILXT_ERR_UNK_VAL);
}

/*-----
 * Check if the data has been converted in GlobalData conversion buffer.
 * This will be the case if the pointer in m_lpData has changed. Check
 * that value here and if it has then copy the data from the conversion
 * buffer pointed to by m_lpData to the field buffer in gd.
 *-----*/
if (m_lpData != pOriginalData)
{
    //----- copy the data from m_lpData to gd->m_lpFldBuf
    IL_STRCPY (gd->m_lpFldBuf, m_lpData);

    //----- reset m_lpData to point to field buffer
    m_lpData = gd->m_lpFldBuf;
}

//----- all done

```

```
    return iRc;  
} //---- CILField::ProcessPut
```

```

#ifndef _ILTIF_INCLUDED
#define _ILTIF_INCLUDED

/*-----
 * Name:      ILTIF.H
 * Purpose:   This file contains all of the structures, defines and includes
 *            used in the ILTIF API for accessing TIF files.
 *
 * Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
 *-----*/

#include "illog.h"
#include "illink.h"

/*-----
 * TIF Phases of Operation (for StartNextPhase calls)
 * - first two #defines are special values used to manage phase transitions
 *   that span address-space switches, e.g Win32/16 or Mac EXE/CodeResource.
 *-----*/
#define TIF_PHASE_PREVIOUS          28
#define TIF_PHASE_NEXT             30
#define TIF_PHASE_SETTING_THINGS_UP 33
#define TIF_PHASE_DONE_SETTING_THINGS_UP 29 // !!! out of sequence !!!
#define TIF_PHASE_LOADING_PREVIOUS_RECORDS 34
#define TIF_PHASE_LOADING_TARGET_RECORDS 35
#define TIF_PHASE_LOADING_SOURCE_RECORDS 36
#define TIF_PHASE_SANITIZING_SOURCE_RECORDS 37
#define TIF_PHASE_CHOOSING_RECORDS 31
#define TIF_PHASE_CONFLICT_RESOLUTION 38

//----- TIFTablePickRecordsForSync() relied on order of next 3 phase numbers
#define TIF_PHASE_UNLOADING_TO_TARGET 39
#define TIF_PHASE_UNLOADING_TO_SOURCE 40
#define TIF_PHASE_UNLOADING_TO_HISTORY 41
#define TIFSYNC_UNLOAD_PHASE_COUNT 3 // the three "unloading_to_xxx" values

/*-----
 * The next 2 definitions are for use of TIF on EXPORT. On EXPORT we clearly
 * are loading from the SOURCE system, but this use of TIF is meant to be
 * simple; no field mapping or source record caching. So we lie and say that
 * we're loading TARGET records.
 *-----*/
#define TIF_PHASE_LOADING_FOR_EXPORT TIF_PHASE_LOADING_TARGET_RECORDS
#define TIF_PHASE_UNLOADING_FOR_EXPORT 42

/*-----
 * The remaining phase definitions are for "pseudo-phases". TIF users
 * call ILTIFStartNextPhase with one of the following phase numbers solely
 * for the sake of getting TIF to add something to TIF.LOG.
 *-----*/
#define TIF_PHASE_FANNING_BEFORE_UNLOAD 51
#define TIF_PHASE_FINISHED_FANNING_BEFORE_UNLOAD 52

/*-----
 * Length of EPOCH string, stored in ILINK.INI and in TIF History Files
 *-----*/
#define TIF_EPOCH_SIZE 16 // YYYYMMDD.HHMMSS\0

#ifdef BUILDING_ILTIF_ITSELF
#include "tif.h"
#endif

// Mechanism Interface globals
typedef struct ILTIF_STRUCT
{
    IL_HANDLE      hstILTIF; // Handle of global structure

#ifdef BUILDING_ILTIF_ITSELF
    ILT_PTIF      pstTIF; // Pointer to global TIF structure
#else
    IL_PANY      pstTIF; // Pointer to global TIF structure
#endif
}

```

```

//----- next 3 members are reusable buffers used to hold field values
ILUT_BUFFER    Field;          // last field value
ILUT_BUFFER    Field2;         // used exclusively by GetMappedField
ILUT_BUFFER    Field3;         // used by MappedFieldChanged and by
                                // GetFieldGuts for Phase20 nested calls
ILUT_BUFFER    TmpBuf;         // for any very localized usage...

BOOLEAN        bTestMode;      // TRUE if running in Test environment

LPSILLOG        lpsILLog;       // Pointer to Logging control block
char            szHistoryName[9]; // basic filename w/o path or extension
char            szHistoryDir[MAX_PATH]; // where history files belong

} ILT_ILTIF;                    // Global data structure

typedef ILT_ILTIF IL_DIST *ILT_PILTIF; // Pointer to above struct

//----- access macros for members of the ILT_ILTIF structure:
#define ILTIF_Field          (ILTR_pILTIF->Field)
#define ILTIF_Field2        (ILTR_pILTIF->Field2)
#define ILTIF_Field3        (ILTR_pILTIF->Field3)
#define ILTIF_TmpBuf        (ILTR_pILTIF->TmpBuf)

#define ILTIF_hstILTIF       (ILTR_pILTIF->hstILTIF)
#define ILTIF_pstTIF        (ILTR_pILTIF->pstTIF)
#define ILTIF_szHistoryName (ILTR_pILTIF->szHistoryName)
#define ILTIF_szHistoryDir  (ILTR_pILTIF->szHistoryDir)
#define ILTIF_szWorkFile    (ILTR_pILTIF->szWorkFile)

// TIF max values
#define TIF_MAX_DISDATE      32          // Maximum display date
#define TIF_MAX_DISTIME      32          // Maximum display time

// TIF Error Defines
#define TIF_ERR_BASE          (-700)
#define TIF_ERR_BAD_PARAM      (TIF_ERR_BASE - 1)
#define TIF_ERR_BAD_FASTSYNC_DELTA (TIF_ERR_BASE - 2)
#define TIF_ERR_MEM            (TIF_ERR_BASE - 3)
#define TIF_ERR_BAD_STATE      (TIF_ERR_BASE - 4)
#define TIF_ERR_BAD_FLDNAME     (TIF_ERR_BASE - 5)
#define TIF_RECERROR           (TIF_ERR_BASE - 6)
#define TIF_EOF                (TIF_ERR_BASE - 7)
#define TIF_ERR_WRONG_PHASE     (TIF_ERR_BASE - 8)
#define TIF_ERR_CANT_CREATE_DIR (TIF_ERR_BASE - 9)
#define TIF_ERR_LOGGING         (TIF_ERR_BASE - 10)
#define TIF_ERR_BAD_MODE        (TIF_ERR_BASE - 11)
#define TIF_ERR_ABSURD_FANOUT   (TIF_ERR_BASE - 12)
//----- next error is when # of ILTIFDefFields calls exceeds number
//----- of field slots allocated in ILTIFCreate call
#define TIF_ERR_PILTIF_IS_NULL (TIF_ERR_BASE - 13)
#define TIF_ERR_BAD_HISTORY_FILE (TIF_ERR_BASE - 14)
#define TIF_ERR_CANT_DELETE_FILE (TIF_ERR_BASE - 15)
#define TIF_ERR_GetTempFileName (TIF_ERR_BASE - 16)
#define TIF_ERR_17 (TIF_ERR_BASE - 17)
#define TIF_NO_MATCH (TIF_ERR_BASE - 18)
#define TIF_ERR_BAD_TIF_ORIGIN (TIF_ERR_BASE - 19)
#define TIF_ERR_BROKEN_SKG (TIF_ERR_BASE - 20)
#define TIF_ERR_PDATA_IS_NULL (TIF_ERR_BASE - 21)
#define TIF_ERR_TARGETID_IN_SOURCE_REC (TIF_ERR_BASE - 22)
#define TIF_ERR_SOURCEID_IN_TARGET_REC (TIF_ERR_BASE - 23)
#define TIF_ERR_BAD_ID_SLOT (TIF_ERR_BASE - 24)
#define TIF_ERR_BROKEN_CIG (TIF_ERR_BASE - 25)
#define TIF_ERR_BAD_CIG_POPULATION (TIF_ERR_BASE - 26)
#define TIF_ERR_NEXT_IS_BAD (TIF_ERR_BASE - 27)
#define TIF_ERR_BAD_OTHER_ORIGIN (TIF_ERR_BASE - 28)
#define TIF_ERR_CANT_JOIN_BAD_CIG (TIF_ERR_BASE - 29)
#define TIF_ERR_CANT_LEAVE_BAD_CIG (TIF_ERR_BASE - 30)
#define TIF_ERR_BAD_NEWMEMBER_ORIGIN (TIF_ERR_BASE - 31)
#define TIF_ILLEGAL_MATCH (TIF_ERR_BASE - 32)
#define TIF_ERR_BAD_ORIGIN (TIF_ERR_BASE - 33)
#define TIF_ERR_HETEROGENEOUS_CIG (TIF_ERR_BASE - 34)
#define TIF_ERR_BAD_CIG_TYPE (TIF_ERR_BASE - 35)
#define TIF_ERR_REMNANT_ISNT_SINGLETON (TIF_ERR_BASE - 36)
#define TIF_ERR_CIG_ISNT_A_PAIR (TIF_ERR_BASE - 37)
#define TIF_ERR_BAD_DATE_OR_TIME_FIELD (TIF_ERR_BASE - 38)

```

```

#define TIF_ERR_BAD_CURRENT_PHASE (TIF_ERR_BASE - 39)
#define TIF_ERR_BAD_NEW_PHASE (TIF_ERR_BASE - 40)
#define TIF_ERR_BAD_RESOLUTION (TIF_ERR_BASE - 41)
#define TIF_ERR_BAD_SMARTMERGE_CIG (TIF_ERR_BASE - 42)
#define TIF_ERR_NOT_YET_IMPLEMENTED (TIF_ERR_BASE - 43)
#define TIF_ERR_BAD_RECORD_NUMBER (TIF_ERR_BASE - 44)
#define TIF_ERR_IMPERTINENT_RECORD (TIF_ERR_BASE - 45)
#define TIF_ERR_IMPOSSIBLE_OUTCOME (TIF_ERR_BASE - 46)
#define TIF_ERR_BAD_FIELD_TYPE (TIF_ERR_BASE - 47)
#define TIF_ERR_TODO_RANGE_CHANGED (TIF_ERR_BASE - 48)
#define TIF_ERR_CANT_FIND_ROOT_FIELD (TIF_ERR_BASE - 49)
#define TIF_SKIP_THIS_RECORD (TIF_ERR_BASE - 50)
#define TIF_ERR_WEIRD_OUTCOME (TIF_ERR_BASE - 51)
#define TIF_SKIP_FAIL_RANGE (TIF_ERR_BASE - 52)
/*-----
 * next 3 errors can only arise when using TIF for APPOINTMENTS, with
 * Appointment Overlap Checking enabled. These errors are reported
 * if field definitions don't meet the Overlap Checker's requirements.
 *-----*/
#define TIF_ERR_MISSING_OVERLAP_FIELD (TIF_ERR_BASE - 53)
#define TIF_ERR_RELATIVE_START_DATE (TIF_ERR_BASE - 54)
#define TIF_ERR_RELATIVE_START_TIME (TIF_ERR_BASE - 55)
#define TIF_PSTTIF_IS_NULL (TIF_ERR_BASE - 56)
#define TIF_ERR_CANT_GET_DLL_HANDLE (TIF_ERR_BASE - 57)
#define TIF_SKIP_AND_DONT_LOG_THIS_RECORD (TIF_ERR_BASE - 58)
#define TIF_ERR_BAD_SMARTMERGE_PHASE (TIF_ERR_BASE - 59)
#define TIF_ERR_BAD_SYNC_UNLOAD_PHASE (TIF_ERR_BASE - 60)
#define TIF_ERR_FIELD_LIST_HAS_CHANGED (TIF_ERR_BASE - 61)
#define TIF_ERR_RECORD_ZERO_TOO_SMALL (TIF_ERR_BASE - 62)
#define TIF_ERR_CANT_FAN_THIS_OUTCOME (TIF_ERR_BASE - 63)
#define TIF_ERR_UTH_LOOPING (TIF_ERR_BASE - 64)
#define TIF_ERR_NON_UNIQUE_SOURCEID (TIF_ERR_BASE - 65)
#define TIF_ERR_NON_UNIQUE_TARGETID (TIF_ERR_BASE - 66)
#define TIF_ERR_BAD_SYNC_CR_OPTION (TIF_ERR_BASE - 67)
#define TIF_ERR_BAD_CR_ACTION (TIF_ERR_BASE - 68)
#define TIF_ERR_BAD_SYNC_CR_ACTION (TIF_ERR_BASE - 69)
#define TIF_ERR_BAD_CIG_TYPE_AFTER_ILCR (TIF_ERR_BASE - 70)
#define TIF_ERR_NULLPTR_FOR_SUBSTITUTION (TIF_ERR_BASE - 71)
#define TIF_ERR_TOO_MANY_NAMEHASH_COLLISIONS (TIF_ERR_BASE - 72)
#define TIF_ERR_CANT_WRITE_SYNC_INI_PARM (TIF_ERR_BASE - 73)
#define TIF_ERR_TOO_MANY_NAMES_UNUSABLE (TIF_ERR_BASE - 74)
#define TIF_ERR_CANT_DELETE_HISTORY_FILE (TIF_ERR_BASE - 75)
#define TIF_ERR_CANT_RENAME_HISTORY_FILE (TIF_ERR_BASE - 76)
#define TIF_ERR_BAD_MSGBOX_RETURN_CODE (TIF_ERR_BASE - 77)
#define TIF_ERR_BAD_ILTR_NSYNCHRONIZE (TIF_ERR_BASE - 78)
#define TIF_ERR_CIG_ISNT_A_TRIPLE (TIF_ERR_BASE - 79)
#define TIF_ERR_CANT_COPY_HISTORYFILE (TIF_ERR_BASE - 80)
#define TIF_ERR_BAD_PHASE_FOR_ACCEPT_OUTCOME (TIF_ERR_BASE - 81)
#define TIF_ERR_FIELD_VALUE_TOO_BIG (TIF_ERR_BASE - 82)
#define TIF_ERR_BAD_PUT_RECORD_OPTION (TIF_ERR_BASE - 83)
#define TIF_ERR_BAD_PHASE_FOR_PUT_RECORD (TIF_ERR_BASE - 84)
#define TIF_ERR_WRONG_FILE_FORMAT_VERSION (TIF_ERR_BASE - 85)
#define TIF_ERR_BAD_GETFIELD_NWHICH (TIF_ERR_BASE - 86)
#define TIF_ERR_87 (TIF_ERR_BASE - 87)
#define TIF_ERR_88 (TIF_ERR_BASE - 88)
#define TIF_ERR_ABNORMAL (TIF_ERR_BASE - 89)
#define TIF_ERR_CANT_FAN (TIF_ERR_BASE - 90)
#define TIF_ERR_BROKEN_FIG (TIF_ERR_BASE - 91)
#define TIF_ERR_INIT_FAILURE (TIF_ERR_BASE - 92)
#define TIF_ERR_ILFldTypeInvertedLookup (TIF_ERR_BASE - 93)
#define TIF_ERR_ILFldTypeEx (TIF_ERR_BASE - 94)
#define TIF_ERR_95 (TIF_ERR_BASE - 95)
#define TIF_ERR_CANT_READ_FROM_NULL_RECORD (TIF_ERR_BASE - 96)
#define TIF_ERR_97 (TIF_ERR_BASE - 97)
#define TIF_ERR_ABSURD_FIELD_LENGTH (TIF_ERR_BASE - 98)
#define TIF_ERR_ABSURD_FIELD_OFFSET (TIF_ERR_BASE - 99)
#define TIF_ERR_FIELD_EXTENDS_BEYOND_RECORD_END (TIF_ERR_BASE - 100)
#define TIF_ERR_101 (TIF_ERR_BASE - 101)
/*-----
 * The following error is returned by STUBs for various sync-specific
 * functions when TIF DLL is built with "NOSYNCPORT=".
 *-----*/
#define TIF_ERR_NOSYNCPORT (TIF_ERR_BASE - 102)

#define TIF_ERR_EMPTY_FIG (TIF_ERR_BASE - 103)

```



```

#define TIF_ERR_NO_SUBTYPE_FIELD (TIF_ERR_BASE - 104)
#define TIF_ERR_NO_SUBTYPE_VALUE (TIF_ERR_BASE - 105)
#define TIF_ERR_BAD_FIELD_NUM (TIF_ERR_BASE - 106)
#define TIF_ERR_KSTRUCT_SIZE_MISMATCH (TIF_ERR_BASE - 107)
#define TIF_ERR_108 (TIF_ERR_BASE - 108)
#define TIF_ERR_DUP_DISTING_FIELD (TIF_ERR_BASE - 109)

// TIF information default values
#define TIF_NOTSET -1 // Offset not set to value yet

// TIF defines for getting field data
#define TIF_ORIGINAL 1 // You want the original data
#define TIF_CURRENT 2 // You want the current data
#define TIF_AUTO 3 // Auto determine cur or orig
#define TIF_SOURCE_CACHE 14 // INTERNAL ONLY: get from Source Cache
#define TIF_FANNING_ADJ 28 // INTERNAL ONLY: for ILTR\REPEAT.C
#define TIF_NESTED_CALL 63 // recursion indicator!!

// TIF Default value defines
#define TIF_BOOL_DEFAULT ILTR_BOOLEAN_FALSE // Boolean field
#define TIF_NUMBER_DEFAULT "0" // Number field
#define TIF_TIME_DEFAULT "0000" // Time field

// Defines for record comparison during conflict resolution
#define TIF_DUPLICATE 1 // Records are duplicate
#define TIF_CONFLICT 2 // Record in conflict
#define TIF_NOCONFLICT 3 // Two different records

// Defines for SmartMerge reconciliation options
#define TIF_REC_OFF -1 // reconciliation off
#define TIF_REC_NONE ILX_OPT_NONE // None reconciliation
#define TIF_REC_IGNORE ILX_OPT_IGNORE // Ignore conflicts
#define TIF_REC_ADD ILX_OPT_INSERT // Add Conflicts
#define TIF_REC_UPDATE ILX_OPT_UPDATE // Update Conflicts
#define TIF_REC_DELETE ILX_OPT_DELETE // Delete Conflicts
#define TIF_REC_REPLACE ILX_OPT_REPLACE // Replace Conflicts
#define TIF_REC_NOTIFY ILX_OPT_NOTIFY // Notify Conflicts

/*-----
 * Defines for Synchronization Conflict Resolution Options
 * Note that ADD_ACROSS means that in the context of a single conflict
 * we start out with 1 record in each app, and end up with 2 in each.
 *-----*/
#define TIFSCRO_LEAVE_UNRESOLVED ILX_OPT_IGNORE
#define TIFSCRO_ADD_ACROSS ILX_OPT_INSERT
#define TIFSCRO_NOTIFY ILX_OPT_NOTIFY
#define TIFSCRO_TARGET_WINS ILX_OPT_ACCEPT_1
#define TIFSCRO_SOURCE_WINS ILX_OPT_ACCEPT_2

#define TIF_PUT_SIMPLE_TEXT 9999 // for TIFRecordAddFieldValue call

/*-----
 * ILTIF Outcomes
 *
 * WARNING: Tiftable.cpp requires that all outcome codes fit in a single byte
 *-----*/
#define ILTIF_OUTCOME_LEAVE_ALONE 0x00000001
#define ILTIF_OUTCOME_ADD 0x00000002
#define ILTIF_OUTCOME_DELETE 0x00000004
#define ILTIF_OUTCOME_REPLACE 0x00000008
#define ILTIF_OUTCOME_UPDATE 0x00000010
#define ILTIF_OUTCOME_IGNORE 0x00000020
#define ILTIF_OUTCOME_LEAVE_DELETED 0x00000040

#define ILTIF_FIRST_TIER_OUTCOME_MASK 0x0000007F

/*-----
 * The TIFPositionToNextRecord function filters out obsoleted TARGET items,
 * but it does not filter out obsoleted SOURCE items, since we may want to
 * log them. (When source items conflict with one another, and
 * user chooses to use one source item to update or replace another, the
 * victim is "obsoleted" and we write "IGNORED" in the xlate log.)
 *-----*/
#define ILTIF_OUTCOME_OBSOLETED 0x00000080

```

```

//--- the following special outcome code is only used by ILTIFFanItem
#define ILTIF_OUTCOME_FANNED      0x00000100L

/*-----
 * The following bits are OR'd together with the above OUTCOME bits to
 * tell a FastSync translator that a "delta" has been fully dealt with.
 *-----*/
#define ILTIF_OUTCOME_DELTA_ACK    0x00000200L

/*-----
 * ILTIFFGetOutcome uses the following mask to make sure no "weird"
 * outcome bits are handed out to the users of ILTIF.
 *-----*/
#define ILTIF_EXPECTED_OUTCOMES    0x000002FF

//---- bit masks for input to ILTIFFeatureSet
#define TIF_DISABLE_FAST_SYNC      0x00000001L
#define TIF_DISABLE_SYNC_BY_ID     0x00000002L

/*-----
 * definitions for extra field attributes defined by TIF
 * use these bits in pExtraAttributes[0], for ILTIFFDefFieldEx() call.
 *-----*/
#define TIFEA_IS_MAPPED             0x00000000
#define TIFEA_ISNT_MAPPED           0x00000001
#define TIFEA_AUTO_FILLIN           0x00000002
#define TIFEA_FIRST_LINE_MAPPED     0x00000004
#define TIFEA_KEY_DATE_FIELD        0x00000008 // used internally

/*-----
 * definitions for logging in high-level ILTIF environment
 *-----*/
#define ILTIFFLOG (ILTR_pILTIF->lpsILLog)
#define ILTIFFlogsz(s)             ILLOG_logsz      (ILTIFLOG,ILLOG_ALWAYS,s)
#define ILTIFFlogszsz(s,t)         ILLOG_logszsz    (ILTIFLOG,ILLOG_ALWAYS,s,t)
#define ILTIFFlogsz3(s,t,u)        ILLOG_logsz3     (ILTIFLOG,ILLOG_ALWAYS,s,t,u)
#define ILTIFFlogsz3ul(s,t,u,v)    ILLOG_logsz3ul   (ILTIFLOG,ILLOG_ALWAYS,s,t,u,v)
#define ILTIFFlogszul(s,t)         ILLOG_logszul    (ILTIFLOG,ILLOG_ALWAYS,s,t)
#define ILTIFFlogszszul(s,t,u)     ILLOG_logszszul  (ILTIFLOG,ILLOG_ALWAYS,s,t,u)
#define ILTIFFlogszulul(s,t,u)     ILLOG_logszulul  (ILTIFLOG,ILLOG_ALWAYS,s,t,u)
#define ILTIFFlogszul3(s,t,u,v)    ILLOG_logszul3   (ILTIFLOG,ILLOG_ALWAYS,s,t,u,v)
#define ILTIFFlogsz2ul2(s,t,u,v)   ILLOG_logsz2ul2  (ILTIFLOG,ILLOG_ALWAYS,s,t,u,v)

#endif

```

```

#ifndef _ILTIFEN_INCLUDED
#define _ILTIFEN_INCLUDED

/*-----
 * Name:      ILTIFEN.H
 * Purpose:  ILTIF Function Prototypes
 *
 * Author:   Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
 *-----*/

#ifdef __cplusplus
extern "C" // everything callable as C functions
{
#endif

#ifdef BUILDING_TIF_AS_STATIC_LIB

#define TIF_DLL_ENTRYPOINT int IL_DECL

#else

#define TIF_DLL_ENTRYPOINT DLLEXPORT int IL_DECL EXP

#endif

/*----- ILTIFCreate is a "classic" entrypoint, which ignores its 2nd arg
//-----and simply calls ILTIFInit. Suggestion: call ILTIFInit instead.
TIF_DLL_ENTRYPOINT ILTIFCreate
( ILTR_PTRANSL tr,
  IL_HINST hObsoleteUnusedArgument,
  INT32 lFields ); // initial Number of fields
// (if not enough, TIF will grow it)

//----- This is the very first TIF function that you should call
//----- SmartMerge apps may call ILTIFCreate, which calls this function.
TIF_DLL_ENTRYPOINT ILTIFInit
( ILTR_PTRANSL tr,
  IL_PSTR lpszTIFFilename,
  INT32 lFields ); // initial Number of fields
// (if not enough, TIF will grow it)

/*-----
 * ILTIFSyncInit and ILTIFSyncFinishUpAndClose
 *
 * high-level entrypoints for synchronization -- in tifsync2.cpp
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFSyncInit ( ILTR_PTRANSL tr,
                                   IL_PSTR szSourceFile,
                                   IL_PSTR szTargetFile );

TIF_DLL_ENTRYPOINT ILTIFSyncFinishUpAndClose (ILTR_PTRANSL tr);

//----- For Debugging, create formatted dump of
//----- the CURRENTLY OPEN TIF file
TIF_DLL_ENTRYPOINT ILTIFDump
( ILTR_PTRANSL tr );

//----- Returns how many fields are currently defined in TIF
TIF_DLL_ENTRYPOINT ILTIFHowManyField
( ILTR_PTRANSL tr,
  INT32 *lNumOfFlds ); // Number of fields returned

//----- Returns field name given an index
TIF_DLL_ENTRYPOINT ILTIFGetFieldName
( ILTR_PTRANSL tr,
  INT32 lFldNdx, // Index of the desired field
  IL_PSTR szFldName ); // Field name returned

//----- Define a field in the TIF database.
TIF_DLL_ENTRYPOINT ILTIFDefField
( ILTR_PTRANSL tr,
  IL_PSTR szFldName, // Field Name
  INT32 lFldSize, // Field size

```

```

    char IL_DIST *pFldType,          // ptr to 1 char Field Type
    IL_PSTR szFormat,                // Field format
    ILTB_ATTRIB FieldAttributes,
    IL_PSTR szDefault );            // Default Value for field

//----- Define a field in TIF, with EXPANDED capabilities
TIF_DLL_ENTRYPOINT ILTIFDefFieldEx
( ILTR_PTRANSL tr,
  IL_PSTR szFldName,                // Field Name
  INT32 lFldSize,                  // Field size
  char IL_DIST *pFldType,          // ptr to 1 char Field Type
  IL_PSTR szFormat,                // Field format
  ILTB_ATTRIB FieldAttributes,
  INT32 *pExtraAttributes,         // IN: array of 1 or more extra longs
                                     // use TIFEA_ #defines in iltif.h...
  IL_PSTR szDefault );            // Default Value for field

TIF_DLL_ENTRYPOINT ILTIFDefFieldN // Tell TIF about a field
( ILTR_PTRANSL tr,
  ILTR_NDX fldnum,                 // field# in ILTR field list
  INT32 ExtraAttributes );

//----- Get a field definition from TIF or from ILTR
TIF_DLL_ENTRYPOINT ILTIFGetFieldAttributes
( ILTR_PTRANSL tr,
  IL_PSTR szFldName,              // IN: Field Name
  INT32 IL_DIST *pMaxLength,      // OUT: Max Field size (or 0 for unlimited)
  char IL_DIST *pFieldType,       // OUT: FldType (ILX_TYPE_XXX)
  ILTB_ATTRIB IL_DIST *pFieldAttributes ); // OUT: Field Attribute bits

//----- Do a standard fill of the TIF mechanism fields, by calling
//----- ILTIFDefField once for every field returned by ILGetField
TIF_DLL_ENTRYPOINT ILTIFFillFields
( ILTR_PTRANSL tr,                // Fill all ILTIF fields from ILIF
  BOOL16 bMappedOnly,            // Fill in only mapped fields?
  BOOL16 bIDNeeded );            // Is a record ID field needed?

/*-----
* ILTIFPutField:
* Place data into a field in the current record. This function assumes
* that character data uses endpoint-specific character encodings. Thus
* all non-binary fields are passed through Character Mapping, which
* includes mapping endpoint-specific EOL character sequences to
* ILTR_EOS_CHAR (0xFF).
*
* For non-binary fields the value supplied as 'lFldSize' helps make
* the debugLog look good, but otherwise it is NOT respected. TIF
* re-computes the Field Length by calling 'strlen'.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFPutField
( ILTR_PTRANSL tr,                // tr struct
  IL_PSTR szFldName,              // Field name
  IL_PANY pFldData,               // Data for the field
  INT32 lFldSize );              // Size of the data

/*-----
* Name:      ILTIFPutFieldEx
* Extended version of ILTIFPutField, called from ILTR\fldput.c, allows
* for 'SpecialFanningAdjustment' calls and control over Character Mapping.
* Called from ILTR\fldput.c\ILTRPutField, with bMapCharsIfNonBinary=FALSE,
* when char mapping has already been done by the ILTRPutField function.
*
* For non-binary fields the value supplied as 'lFldLen' helps make
* the debugLog look good, but otherwise it is NOT respected. TIF
* re-computes the Field Length by calling 'strlen'.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFPutFieldEx
( ILTR_PTRANSL tr,                // tr struct
  IL_PSTR szFldName,              // Field name
  IL_PANY pFldData,               // Data for the field
  INT32 lFldLen,                  // Length of the data
  BOOLEAN bSpecialFanningAdjustment, // TRUE for Date Adj. when fanning
  BOOLEAN bMapCharsIfNonBinary ); // TRUE to have char mapping done
                                     // for non-binary text fields

```

```

//----- Sometimes Map Fields, and Write the record to the TIF.
TIF_DLL_ENTRYPOINT ILTIFPutRecord
( ILTR_PTRANS� tr );

//----- Create clean slate, ready for 'PutField' calls to fill'er up.
TIF_DLL_ENTRYPOINT ILTIFInitRecord
( ILTR_PTRANS� tr );

//----- Next entrypoint is now IDENTICAL to ILTIFPutRecord
TIF_DLL_ENTRYPOINT ILTIFWriteRecord
( ILTR_PTRANS� tr );

//----- arrange for a copy of the Current Record to be used
//----- as target of any subsequent PutField calls, and
//----- as record to use for subsequent ILTIFPutRecord call.
TIF_DLL_ENTRYPOINT ILTIFCloneRecord
( ILTR_PTRANS� tr );

/*-----
* Name:      ILTIFEndLoad
* Purpose: Tells TIF that we're done loading, ready to start unloading.
*          When and ILX_V3 style TARGET translator is using TIF, this
*          call causes TIF to Analyze and Resolve Conflicts.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFEndLoad
( ILTR_PTRANS� tr );

/*-----
* ILTIFStartNextPhase
*
* Signals beginning of a new phase, and indicates end of previous phase
*
* use TIF_PHASE_XXX definitions from ILTIF.H
*-----*/
TIF_DLL_ENTRYPOINT ILTIFStartNextPhase
( ILTR_PTRANS� tr,
  INT16 phase );

/*-----
* Name:      ILTIFSetPositionAboveTopRecord
* Purpose: Set Position such that a subsequent ILTIFNextRecord or
*          ILTIFReadNextRecord call will get the FIRST record that
*          pertains to the current UNLOADING PHASE.
* NOTE: returns SUCCESS even if there are ZERO records to unload.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFSetPositionAboveTopRecord ( ILTR_PTRANS� tr );

// ILTIFTopRecord is OBSOLETE: use ILTIFSetPositionAboveTopRecord instead

//----- Goto next record from the TIF.
TIF_DLL_ENTRYPOINT ILTIFNextRecord
( ILTR_PTRANS� tr );

/*-----
* Name:      ILTIFGotoRecord
* Purpose: Goto a specified record in the TIF, verifying that the
*          specified record exists and pertains to the current UNLOADING
*          PHASE.
* NOTE: Code that uses ILTIF should not make assumptions about the
*        numbering of TIF records. Record numbers passed to this
*        function should NOT be generated by users of ILTIF; rather
*        the numbers should be gotten by calling ILTIFRecordNum. A
*        typical usage pattern is to do one pass over the TIF, using the
*        [Read]NextRecord to iterate; using ILTIFRecordNum to get
*        record numbers, storing them, maybe filtering or sorting them,
*        then doing a second pass which iterates through the stored
*        record numbers, using ILTIFGotoRecord for positioning.
*
*        ILTIF users who don't need to do filtering or sorting probably
*        don't need to use this function at all.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFGotoRecord
( ILTR_PTRANS� tr,

```

```

        INT32 lRecNum );                // record number to position on

//----- Return the current record number
TIF_DLL_ENTRYPOINT ILTIFRecordNum
( ILTR_PTRANS� tr,
  INT32 *lRecNum );                // Pointer to buffer to return rec num

//----- Read next record from the TIF.
TIF_DLL_ENTRYPOINT ILTIFReadRecord
( ILTR_PTRANS� tr );

//----- Goto next record from the TIF and read it.
TIF_DLL_ENTRYPOINT ILTIFReadNextRecord
( ILTR_PTRANS� tr );

//----- Goto specified record from the TIF and read it.
TIF_DLL_ENTRYPOINT ILTIFReadRecordNum
( ILTR_PTRANS� tr,
  INT32 lRecNum );                // Record to retrieve

/*-----
 * Get field value for given field, into a buffer owned by TIF, and
 * return Length == STRLEN(text)+1 for text fields, or "Exact Length"
 * for binary fields. Field values are NOT truncated.
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFGetField
( ILTR_PTRANS� tr,
  IL_PSTR szFldName,                // Field Name
  int nWhich,                      // Current, original, or AUTO
  INT32 *lField,                   // OUT: length of value returned
  IL_PANY *pField );               // Buffer for the field data

/*-----
 * Get field value for given field and copy it into the caller's buffer.
 * Return Length == STRLEN(text) (NOT STRLEN()+1) for text fields,
 * or "Exact Length" for binary fields. Field values ARE truncated as
 * necessary, and ILTR_ERR_TRUNC error is returned when that happens.
 * (Called from within the ILTR\fldget.c\GetField function.)
 *
 * NOTE: For text, the IN value of *pLength is max size INCLUDING null
 *       terminator, but the OUT value of *pLength is STRLEN(text)
 *       -- not including null.
 *
 * So to GetAndCopy the value "1995", you must supply *pLength >= 5,
 * and when the call completes you'll have *pLength == 4.
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFGetAndCopyField
( ILTR_PTRANS� tr,
  IL_PSTR FieldName,
  int nWhich,                      // Current, original, or AUTO
  INT32 *pLength,                  // IN=MAX / OUT=ACTUAL length of value
  IL_PANY *pBuffer );              // Buffer for the field data

//----- Set the data translation option for TIF mechanism
TIF_DLL_ENTRYPOINT ILTIFSetDataConv
( ILTR_PTRANS� tr,
  int nOption );                  // Data translation option

//----- Set the TIF reconciliation option
TIF_DLL_ENTRYPOINT ILTIFSetReconcile
( ILTR_PTRANS� tr,
  int nOption );                  // Reconciliation option

/*-----
 * Record Outcome Functions:
 *
 * These functions are used, during an UNLOADING PHASE, to get the
 * OUTCOME flags for a given record. The 'GetOutcome'
 * function returns a long word full of flag bits; the other
 * functions in this group call 'GetOutcome', then check
 * for various bit flags in the flag word.
 *
 * Functions are:
 *
 * ILTIFGetOutcome

```

```

*      ILTIFRecordAdded
*      ILTIFRecordChanged
*      ILTIFRecordDeleted
*      ILTIFRecordReplaced
*
*      ILTIFFieldChanged
*
*      All of the Record-level functions work from in-memory TIF
*      knowledge, so they need not be preceded by a ReadRecord call.
*
*      Unlike the Record-level functions, the ILTIFFieldChanged function
*      must be preceded by a ReadRecord call.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFGetOutcome
( ILTR_PTRANSL tr, INT32 IL_DIST *pOutcome );

TIF_DLL_ENTRYPOINT ILTIFRecordAdded    ( ILTR_PTRANSL tr );
TIF_DLL_ENTRYPOINT ILTIFRecordChanged  ( ILTR_PTRANSL tr );
TIF_DLL_ENTRYPOINT ILTIFRecordDeleted  ( ILTR_PTRANSL tr );
TIF_DLL_ENTRYPOINT ILTIFRecordReplaced ( ILTR_PTRANSL tr );
TIF_DLL_ENTRYPOINT ILTIFFieldChanged   ( ILTR_PTRANSL tr,
                                          IL_PSTR szFldName );

/*-----
* Functions for Accepting or Rejecting a Record Outcome
*
*      When a translator is unloading records from TIF, TIF assumes that
*      the prescribed record outcomes will be effected by the translator.
*
*      As long as this assumption holds true, nothing special need be
*      done. But it is a good practice, especially when doing
*      Synchronization, for the translator to call ILTIFAcceptOutcome
*      for each record. Currently this is REQUIRED for INSERT operations
*      done during synchronization if the translator wants to supply a
*      unique ID for the newly-created item, and for UPDATE operations
*      that cause a new unique ID to be assigned by the application.
*
*      NOTE that TIF does NOT allow for the possibility that a LEAVE_ALONE
*      outcome might result in assignment of a new unique ID. If any
*      application has that behavior then unique IDs are UNUSABLE for
*      IntelliLink synchronization, and the translator must not tell TIF
*      that Unique IDs exist!!
*
*      Although TIF allows for the possibility that a sync-driven UPDATE
*      may force assignment of a new ID, that is probably unusual
*      behavior, and if the application is known to assign new IDs when
*      users make changes then that apps IDs are UNUSABLE for
*      IntelliLink Synchronization.
*
*      On the other hand if a translator finds that it cannot put a record
*      outcome into effect (e.g. it cannot add a record), it has two
*      choices:
*
*      1. it can treat the failure as a FATAL error which aborts the
*         entire operation, or
*
*      2. it can simply tell TIF that the outcome for this particular
*         record is rejected, then continue processing.
*
*      For option #2 it must call ILTIFRejectOutcome. At this point in
*      time no values are defined for the 2nd argument to this function.
*-----*/

/*-----
* ILTIFAcceptOutcome
* -- for 2nd arg, pass NULL (not "") if you don't have a New Unique ID
*    to tell TIF about.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFAcceptOutcome (ILTR_PTRANSL tr, IL_PSTR newUniqueID);

/*-----
* ILTIFRejectOutcome
*-----*/
TIF_DLL_ENTRYPOINT ILTIFRejectOutcome (ILTR_PTRANSL tr, int errorCode);

```



```

//----- Shut Down the TIF mechanism and delete the TIF file
TIF_DLL_ENTRYPOINT ILTIFClose
( ILTR_PTRANSL tr );

/*-----
* Name:      ILTIFCloseFileTemporarily -- for xlators to call
* Purpose: Close-the TIF workfile file at translation phase transition
*-----*/
TIF_DLL_ENTRYPOINT ILTIFCloseFileTemporarily (ILTR_PTRANSL tr);

/*-----
* Name:      ILTIFCloseFileInitially -- for engine to call
* Purpose: Close the TIF file without updating it or shutting things down
*-----*/
TIF_DLL_ENTRYPOINT ILTIFCloseFileInitially (ILTR_PTRANSL tr);

/*-----
* Name:      ILTIFReopenFile
* Purpose: Re-open a TIF file that was previously closed by calling
*           ILTIFCloseFileTemporarily
*-----*/
TIF_DLL_ENTRYPOINT ILTIFReopenFile (ILTR_PTRANSL tr);

/*-----
* Status Bar Functions: (& old names for same...)
*
* ILTIFBeginStatusBar - Initialize and display the status bar
* ILTIFUpdateStatusBar - Increment the status bar
* ILTIFEndStatusBar - Take down and clean up the status bar
*-----*/
TIF_DLL_ENTRYPOINT ILTIFBeginStatusBar
( ILTR_PTRANSL tr,
  IL_PSTR szTheMsg,          // The message
  INT32 lNumRecs );          // Number of updates

TIF_DLL_ENTRYPOINT ILTIFUpdateStatusBar
( ILTR_PTRANSL tr );

TIF_DLL_ENTRYPOINT ILTIFEndStatusBar
( ILTR_PTRANSL tr );

/*-----
* Old Names for the Status Bar Functions:
*-----*/
TIF_DLL_ENTRYPOINT ILTIFBeginStatus
( ILTR_PTRANSL tr,
  IL_PSTR szTheMsg,          // The message
  INT32 lNumRecs );          // Number of updates

TIF_DLL_ENTRYPOINT ILTIFUpdateStatus
( ILTR_PTRANSL tr );

TIF_DLL_ENTRYPOINT ILTIFEndStatus
( ILTR_PTRANSL tr );

//----- Unload all fields in all records from ILTIF to ILIF
TIF_DLL_ENTRYPOINT ILTIFUnloadToILIF
( ILTR_PTRANSL tr );

//----- Return the number of records in the TIF mechanism
TIF_DLL_ENTRYPOINT ILTIFHowManyRecords
( ILTR_PTRANSL tr,
  INT32 *lNumOfRecs );          // Pointer to number of records

/*-----
* Name:      ILTIFDontSyncByID
* Purpose: Tells the Synchronization Engine not to use Unique IDs in doing
*           synchronization.
*
* NOTE:      calling this function is equivalent to calling
*           ILTIFFeatureSet (tr, 0, TIF_DISABLE_SYNC_BY_ID);
*
* Comments: This call turns off usage of unique IDs for sync, but TIF still
*           does ordinary storage and retrieval of unique IDs, if unique ID
*           fields are defined.
*-----*/

```



```

*
*      This function is only needed when a translator that has a field
*      in its field list called _UniqueID wants TIF to do KeyField-based
*      synchronization rather than uniqueID-based synchronization.
*
*      For a synchronization job involving translators ILXAAA and ILXBBB,
*      the field lists of one or the other or both or neither of ILXAAA
*      and ILXBBB may have _UniqueID fields.  If both have _UniqueID
*      fields, then if ILXAAA calls ILTIFDontSyncByID the sync engine
*      will do KeyField-based correlation on the ILXAAA side of things,
*      but correlation on the ILXBBB side will still be done by Unique ID.
*
*      TIF requires that this function be called either during the
*      load-from-target phase or during the load-from-source phase.
*
*      However TIF does not apply any other restrictions on when this
*      function may be called.  Unless wacky counter-arguments suggest
*      otherwise, a translator should call ILTIFDontSyncByID before
*      putting any records into TIF.  Suggestion: call it in your
*      BEGIN routine when ILTR_direction == ILTR_EXPORT.
*
*      Author: David Boothby, Copyright (c) IntelliLink Corporation.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFDontSyncByID (ILTR_PTRANSL tr);

/*-----
* Name:      ILTIFFeatureSet
* Purpose: Tells TIF to turn specified feature bits ON or OFF.
*
* Comments: This function has a general-purpose interface, but initially
*           it's only use is to disable FastSync and/or Sync-by-UniqueID
*           for the current phase.
*
* Author: David Boothby, Copyright (c) IntelliLink Corporation.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFeatureSet ( ILTR_PTRANSL tr,
                                     UINT32 ulTurnOnBits,
                                     UINT32 ulTurnOffBits );

/*-----
* ILTIFItemIsRecurring
*-----*/
TIF_DLL_ENTRYPOINT ILTIFItemIsRecurring (ILTR_PTRANSL tr);

/*-----
* ILTIFFanItem
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFanItem (ILTR_PTRANSL tr, int maxFanCount);

/*-----
* ILTIFNewEpoch -- record new Start Of Epoch date&time stamp in ILINK.INI
*-----*/
TIF_DLL_ENTRYPOINT ILTIFNewEpoch
( int sysid,          // system number from ilsysids.h
  IL_PSTR szWorkDir,  // directory where IntelliLink lives
  IL_PSTR szAppFile,  // name of app file or app status file
  IL_PSTR szEpoch ); // YYYYMMDD.HHMMSS\0

/*-----
* ILTIFRemoveRecord
*
* To implement a 'chooser' function, for SmartMerge, you must call
* ILTIFStartNextPhase (tr, TIF_PHASE_CHOOSING_RECORDS), just before calling
* ILTIFEndLoad; then you can read the incoming (source) records, just as if
* you were in the UNLOAD-to-TARGET phase, and then for any records that you
* want to remove (so that they aren't visible to the conflict resolution
* and unload-to-target phases) you simply call ILTIFRemoveRecord.
*
* If you want to remove the "current" record, pass lRecNum = -1.
*
* For example:
*
*      rc = ILTIFReadNextRecord (tr);
*      rc = ILTIFRemoveRecord (tr, -1);
*

```

```

* If you want to remove an arbitrary record, pass actual recnum.
*
* For example:
*
*         rc = ILTIFFReadNextRecord (tr);
*         rc = ILTIFFRecordNum (tr, &lRecNum);
*         rc = ILTIFFRemoveRecord (tr, lRecNum);
*
* When your 'chooser' function is finished, call ILTIFFEndLoad, etc.
*
* Records that you have 'removed' will show up in TIF.LOG with
* ODD Flags values (the GARBAGE bit is 0x00000001).
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFRemoveRecord (ILTR_PTRANS� tr, INT32 lRecNum);

/*-----
* ILTIFFWipeOutHistory
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFWipeOutHistory (ILTR_PTRANS� tr);

#ifdef __cplusplus
}
#endif

#ifdef BUILDING_ILTIFF_ITSELF
#include "tiffn.h"
#endif

#endif

```

```

#ifndef _TIF_INCLUDED
#define _TIF_INCLUDED

/*-----
 * Name:      TIF.H
 * Purpose: Private definitions used only within ILTIF/TIF.
 *
 * This file should NOT be included by any modules
 * outside of the ILTIF subsystem.
 *
 * NOTE: nobody, not even any of the TIF modules themselves, should
 * directly include this file. It's inclusion is controlled by
 * ILTIF.H.
 *
 * Author: Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
 *-----*/

#include "tifrc.h"
#include "ildfx.h"
#include <assert.h>

#define TIF_ASSERT assert

#define IL_HPSTR char IL_HUGE *
#define IL_USHPSTR unsigned char IL_HUGE *

#ifdef BUILDING_TIF_AS_STATIC_LIB

    extern IL_HINST hXlatorInst;    // see ILTR EXPORT.C
    #define TIF_DLL_InstanceHandle hXlatorInst

#else

    extern IL_HINST TIF_DLL_InstanceHandle; // see LibMain in iltif.cpp

#endif

/*-----
 * Parameters used for sizing reusable field buffers. Initial field
 * buf size is big enough for _repBasic. Weird number is "distinctive".
 *-----*/
#define TIF_INITIAL_FIELD_BUF_SIZE 153
#define TIF_BUF_INC 2048
#define TIF_MAX_RECORD_SIZE 3200000 // arbitrary 3.2 megabyte limit

//----- currently the range of VALID cig types is 1-16 (ct000 isn't valid)
#define TIFSYNC_CIG_TYPE_COUNT 16 // must "agree" with TIFCIG_MAX

//---- when hashing and comparing descriptions, only use first 19 characters
#define TIF19 19

//----- Decoration of UNMAPPED Source Field Names
#define TIF_USFN_PREFIX "\\01"
#define TIF_USFN_FORMAT TIF_USFN_PREFIX "%s"

// Unique ID Field Names recognized by TIF
#define TIF_SOURCEID_FIELDNAME TIF_USFN_PREFIX ILTR_FLD_UNIQUE_ID
#define TIF_TARGETID_FIELDNAME ILTR_FLD_UNIQUE_ID

/*-----
 *
 * The first several records of the TIF File have special uses -- see the
 * definitions of TIF_RECORD_ZERO/ONE/TWO below. Higher-numbered records
 * starting with TIF_FIRST_ITEMNO contain TIF_RECORD_VALUE structs.
 *
 * WARNING:: all file-resident structures must use size-invariant types to
 * allow for hybrid translations (Win32/16) & cross-platform usage of tools.
 *-----*/

// Field Descriptor -- Data Definition for a Single field
typedef struct
{
    INT16 FieldNum;                // zero-based fieldnum of this field
    INT16 RelatedFieldNum;         // zero-based fieldnum of related field

```

```

    char szFldName[ILTR_MAX_FLDNAME+1]; // Field Name w/room for decoration
    INT32 MaxLength;                    // Maximum allowed length for this field
    INT32 MaxMappedLength;              // Max length of SOURCE field to which this
                                        // TARGET field is mapped
    char FieldType;                    // Field Data type
    char szFormat[ILTR_MAX_FLDNAME];    // Format string for translation
    char RelatedFieldName[ILTR_MAX_FLDNAME];
    ILTB ATTRIB FieldAttributes;        // various bit flags (UINT32)
    INT32 ExtraAttributes;              // more attribs, see TIFEA_XXX #defines
    BOOL16 FieldIsAdded;                // Is Assoc field added
    BOOL16 bSourceIDField;
    BOOL16 bTargetIDField;
    INT16 FieldToGetDefaultValueFrom; // set to get default from another fld
    char szDefault[ILTR_MAX_FLDNAME]; // Default value for field
} TIF_FIELD_DESC;                      // Single field descriptor

typedef TIF_FIELD_DESC IL_DIST *TIF_FIELD_DESC_PTR; // ptr to above struct

// FieldList structure -- put in Record Zero of TIF File
typedef struct
{
    INT16 AllocatedFieldCount;
    INT16 ActualFieldCount;
    INT16 FirstOffset;           // OBSOLETE: Offset to 1st byte of 1st field
    TIF_FIELD_DESC pFieldDescs[]; // Array of field descriptors
} TIF_FIELDLIST;

typedef TIF_FIELDLIST IL_DIST *TIF_FIELDLIST_PTR; // Pointer to above struct

/*-----
 * TIF File Format Version -- stored in record zero of the TIF file.
 * When updating Version#, increment the high byte when the size or shape
 * of key structures in the file format change. Increment the low byte
 * when interpretation of elements changes.
 *-----*/
#define TIF_SHAPE 0x53
#define TIF_INTERP 0x07
#define TIF_CURRENT_FILE_FORMAT_VERSION ((0x100 * TIF_SHAPE) + TIF_INTERP)

#define TIF_FIELD_COUNT_INCREMENT 10 // growth step

/*-----
 *
 * Note: The structures below are allocated once per record, and then
 * each time a data is added to the record the structure is
 * reallocated and the offset of the data is stored in the
 * proper field struct. With this scheme a field can be overwritten
 * many times. The new data just gets added to the bottom and
 * the field structure points there. The old data for this field
 * then become dead space.
 *-----*/

// Field Value Header -- the Record Value Struct contains an array of these
typedef struct
{
    INT32 FieldOffset; // Offset of data at bottom of struct
    INT32 lFldSize;    // Size of the field data
} TIF_FIELD_VALUE;    // Info about a single field value

typedef TIF_FIELD_VALUE IL_DIST *TIF_FIELD_VALUE_PTR; // ptr to above struct

// Record Value Structure -- Fixed Size Header Part
typedef struct
{
    INT32 lRecSize; // Total size of record
    INT32 lCommand; // Reserved for future use
}
TIFREC_FIXED_PART;

// Record Value Structure
typedef struct

```

```

{
    TIFREC_FIXED_PART R;
    TIF_FIELD_VALUE    pFieldValues[];    // Array of Field Value structs
}
TIF_RECORD_VALUE;

typedef TIF_RECORD_VALUE IL_DIST *TIF_RECORD_VALUE_PTR; // Record Value Pointer

/*-----
 * Use the following macros to access members of a Record Value Header
 * pointed to by a Record Value Pointer.
 *-----*/
#define TIFREC_SIZE(pr)          (pr->R.lRecSize)
#define TIFREC_COMMAND(pr)      (pr->R.lCommand)

typedef ILX_OPTION TIF_RECONCILIATION_OPTION;

// Outcome Counts Structure, used to gather inputs for OKToProceed
typedef struct
{
    long LeaveAloneCount;
    long AddCount;
    long UpdateCount;
    long ReplaceCount;
    long DeleteCount;
    long IgnoreCount;
}
TIF_OUTCOME_COUNTS;

// Date Range Structure
typedef struct
{
    INT32 lStartDate;
    INT32 lEndDate;
}
TIF_DATERANGE;

/*-----
 * The TIF_KSTRUCT type holds parameters that remain constant for all phases
 * of TIF usage. For Win32/16 mixed mode operation these parameters must
 * be shared (i.e. passed) across the 32/16 boundary.
 * TIF parameters that are NOT stored in TIF_KSTRUCT are either fleetingly
 * meaningful or are reconstructed when we cross the 32/16 boundary.
 *-----*/
typedef struct
{
    INT16 FileFormatVersion;    // (TIF_CURRENT_FILE_FORMAT_VERSION)

    //----- Distinguished Field Indices
    INT16 AlarmFlagFieldNum;
    INT16 AlarmTimeFieldNum;
    INT16 AlarmDateFieldNum;
    INT16 DoneFlagFieldNum;
    INT16 StartDateFieldNum;
    INT16 EndDateFieldNum;
    INT16 StartTimeFieldNum;
    INT16 EndTimeFieldNum;
    INT16 ViewFieldNum;
    INT16 SourceIDFieldNum;
    INT16 TargetIDFieldNum;
    INT16 SourceDeltaFieldNum;
    INT16 TargetDeltaFieldNum;
    INT16 RepBasicFieldNum;
    INT16 RepExclFieldNum;
    INT16 SubTypeFieldNum;
    INT16 SourceSubTypeFieldNum;
    INT16 KeyDateFieldNum;

    /*-----
     * Next flags say whether the translators are FastSync-capable, whether
     * or not the loading done for current job is Fast or Slow.
     *-----*/
    BOOL16 bSourceIsFastSyncAble;
    BOOL16 bTargetIsFastSyncAble;

```

```

/*-----
 * Next flags say whether the loading done for the current job was done
 * Fast or Slow.
 *-----*/
BOOL16 bFastSyncSourceLoad;
BOOL16 bFastSyncTargetLoad;

/*-----
 * Next 2 flags are normally set based on existence of fields called
 * "_UniqueID" in the Target and Source Field Lists. But translators
 * can override standard behavior by calling ILTIFDontSyncByID.
 *-----*/
BOOL16 bSyncUsingTargetIDs;
BOOL16 bSyncUsingSourceIDs;

/*-----
 * Next 3 flags control various behaviors
 *-----*/
BOOL16 bCheckForOverlap;           // for appointments
BOOL16 bReconcileExclusions;       // for recurring items
BOOL16 bReconcileRepBasic;         // for recurring items
BOOL16 bDontSyncDoneTodos;         // ILTR_nSchOpt == ILTR_TODO_NOTDONE

/*-----
 * Date Range info, used for synchronization only
 *-----*/
TIF_DATERANGE RangeOfSync;         // date range requested by user this time
TIF_DATERANGE RangeOfPreviousSync;  // date range requested by user last time
TIF_DATERANGE RangeOfSourceLoad;    // date range used when loading src recs

//---- Date when previous synchronization occurred
INT32 lDateOfPreviousSync;

/*-----
 * Epoch values, used for sync only, and not used for all systems. Used
 * to decide whether a history file is still usable, or if it has been
 * invalidated by some external event, such as a user doing a hard
 * reset of a handheld device. The epoch value stored in a history
 * file is compared with the epoch value stored in ILINK.INI to check
 * for possible invalidation.
 *-----*/
char SourceEpoch[TIF_EPOCH_SIZE]; // date&time stamp (YYYYMMDD.HHMMSS\0)
char TargetEpoch[TIF_EPOCH_SIZE]; // date&time stamp (YYYYMMDD.HHMMSS\0)

/*-----
 * Max fanout counts are kept for source and target translators. See
 * ILRPT.H for definition of the ILTR_FANOUT_MAXIMA structure.
 *-----*/
ILTR_FANOUT_MAXIMA SourceFanoutMaxima; // max count for source app
ILTR_FANOUT_MAXIMA TargetFanoutMaxima; // max count for target app
}
TIF_KSTRUCT;

/*-----
 * TheILT_TIF structure type (typically pointed to by *pstTIF)
 * is the primary control structure for the innards of TIF.
 *-----*/
typedef struct
{
    TIF_KSTRUCT          K;           // "Konstant Stuff"

    IL_HANDLE            hstTIF;      // Handle of global structure

/*-----
 * Next 2 members are for an Array of Field Descriptions used to describe
 * all Target Fields and all UNMAPPED Source Fields.
 *-----*/
    TIF_FIELDLIST_PTR    pFieldList;
    IL_HANDLE            hFieldList;

/*-----
 * Next 2 members are for an Array of Field Descriptions used to describe
 * all MAPPED Source Fields. Populated in Phase 05 (SYNC ONLY).
 *-----*/
}

```

```

TIF_FIELDLIST_PTR      pSourceFieldList;
IL_HANDLE              hSourceFieldList;

INT32                  lCurrentRecNum;    // actual recnum while unloading
INT32                  lOriginalRecNum;    // actual recnum while unloading

ILUT_BUFFER            CurrentRecord;
ILUT_BUFFER            OriginalRecord;
ILUT_BUFFER            FirstRecord;
ILUT_BUFFER            SecondRecord;
ILUT_BUFFER            SourceRecord;      // used for caching in Phase20

ILUT_BUFFER            CurrentField;
ILUT_BUFFER            OriginalField;
ILUT_BUFFER            ExtraField;

ILUT_PBUFFER           pFanBuf;

ILDFX_HNDL             hILDFX_File;       // Handle to ILDFX file
BOOLEAN               bWorkFileIsOpen;

INT32                  TotalRecordCount;   // see TIF_TotalRecordCount
INT32                  PertinentRecordCount; // Number of records for
                                                // current UNLOADING PHASE

INT32                  CurrentRecordNumber; // virtual recnum
INT32                  CurrentRecordOutcome; // outcome for most recent-
                                                // ly read virtual record

TIF_RECONCILIATION_OPTION Resolution;      // res. for 1 conflict
INT32                  AffectedItem;       // record that user has
                                                // chosen to update

BOOLEAN               bSkipLeaveAlones;    // Set for non-total-rebuild unloading
BOOLEAN               bFastSyncLoad;      // Set while doing a "Fast Sync Load"
BOOLEAN               bFastSyncUnload;    // Set while doing a "Fast Sync Unload"
BOOLEAN               bUserCancel;        // Set if user presses CANCEL
LPSILLOG              lpsILLog;           // Logging control block
INT16                 phase;               // phase of TIF operation
INT32                 origin;

ILTR_PTRANS�          tr;                  // yecch! - just in case we ever need it
struct ILTIF_STRUCT IL_DIST *pILTIF;      // pointer to parent structure
                                                // gross & hideous but needed for TIF
                                                // code to access stuff in ILTIF struct
char                  PreviousILTRPhase;   // copy of ILTR_phase
BOOLEAN               RecordHasBeenPutSinceLastGet; // flag maintained by ILTIFReadRecord
                                                // and ILTIFPutRecord, to keep track
                                                // of what's going on during the
                                                // 'Sanitizing Source Records' phase.

int nWhich;           // Original, Current, or Auto (#defines below)

/*-----
 * Next flag is set to TRUE, by TIF, when we're unloading
 * records and we want each unloaded record to be logged in
 * the USER-VISIBLE logfile (xlate.log or il.log).
 *-----*/
BOOLEAN bCurrentlyLoggingAndCountingRecords;
BOOLEAN bNeedPriming; // TRUE until record has been init'd
BOOLEAN bPleaseSaveFanoutMaxima; // TRUE until the Maxima have been saved

char szWorkFile[MAX_PATH]; // Filename for TIF File

ILXTR_SYNC_OPTION nSynchronize; // copy of ILTR_nSynchronize
UINT32 CRPolicy; // Conflict Resolution Policy (copy of ILTR_CRPolicy)

TIF_RECONCILIATION_OPTION ReconciliationOption;

//---- the next 5 localizable strings are used in ILCR
char szRepStartPseudoFldName[ILTR_MAX_FLDNAME+1];
char szRepStopPseudoFldName[ILTR_MAX_FLDNAME+1];
char szRepExPseudoFldName[ILTR_MAX_FLDNAME+1];
char szTrue[20];
char szFalse[20];

```

```

/*-----
 * Next 2 recnum values are used to limit forward scanning of TIF index
 * when unloading. Used to avoid unloading instances generated during
 * unload.
 *
 * Both GoalPostXXX values are set by TIFComputePertinentRecordCount.
 * (This allows for good basic unload behavior even if user
 * never calls ILTIFFHowManyRecords.)
 *
 * The 'GoalPostForNextPass' value is incremented when new records
 * are added during unload, and is copied into GoalPost
 * by ILTIFFHowManyRecords.
 *
 * After any unload procedure that generates new records, such as
 * when CILTranslator::FanBeforeUnload calls ILTIFFanItem or when
 * ILTIFFAcceptOutcome calls CreateOneFigMember, the scan-stopping
 * Goal Post is pushed out IF AND ONLY IF one calls ILTIFFHowManyRecords.
 *-----*/
INT32 GoalPost;
INT32 GoalPostForNextPass;

//---- outcome-counting stuff needed for OKToProceed
BOOLEAN bCountingOutcomes;
TIF_OUTCOME_COUNTS OutcomeCounts;

//---- see TIFTABLE.CPP for explanation of the following table
unsigned char TableAutomatic [TIFSYNC_CIG_TYPE_COUNT]
                             [TIFSYNC_UNLOAD_PHASE_COUNT]
                             [3];

) ILT_TIF;

typedef ILT_TIF IL_DIST *ILT_PTIF;    // Pointer to above struct

typedef ILT_PTIF IL_DIST *PSTTIF_TYPE;    // pointer to pointer to above struct

//----- access macros for members of the above structs
#define TIF_hstTIF          ((*pstTIF)->hstTIF)

#define TIF_pFieldList      ((*pstTIF)->pFieldList)
#define TIF_hFieldList      ((*pstTIF)->hFieldList)

#define TIF_pSourceFieldList ((*pstTIF)->pSourceFieldList)
#define TIF_hSourceFieldList ((*pstTIF)->hSourceFieldList)

#define TIF_lCurrentRecNum   ((*pstTIF)->lCurrentRecNum)           // actual
#define TIF_lOriginalRecNum ((*pstTIF)->lOriginalRecNum)           // actual

#define TIF_CurrentRecord    ((*pstTIF)->CurrentRecord)
#define TIF_pCurrentRecord  ( (TIF_RECORD_VALUE_PTR)             \
    (TIF_CurrentRecord.pBuffer) )
#define TIF_OriginalRecord   ((*pstTIF)->OriginalRecord)
#define TIF_pOriginalRecord  ( (TIF_RECORD_VALUE_PTR)             \
    (TIF_OriginalRecord.pBuffer) )
#define TIF_FirstRecord      ((*pstTIF)->FirstRecord)
#define TIF_pFirstRecord     ( (TIF_RECORD_VALUE_PTR)             \
    (TIF_FirstRecord.pBuffer) )
#define TIF_SecondRecord     ((*pstTIF)->SecondRecord)
#define TIF_pSecondRecord    ( (TIF_RECORD_VALUE_PTR)             \
    (TIF_SecondRecord.pBuffer) )
#define TIF_SourceRecord     ((*pstTIF)->SourceRecord)
#define TIF_pSourceRecord    ( (TIF_RECORD_VALUE_PTR)             \
    (TIF_SourceRecord.pBuffer) )

#define TIF_CurrentField     ((*pstTIF)->CurrentField)
#define TIF_pCurrentField    ((*pstTIF)->CurrentField.pBuffer)

#define TIF_OriginalField     ((*pstTIF)->OriginalField)
#define TIF_pOriginalField    ((*pstTIF)->OriginalField.pBuffer)

#define TIF_ExtraField        ((*pstTIF)->ExtraField)
#define TIF_pExtraField       ((*pstTIF)->ExtraField.pBuffer)

#define TIF_pFanBuf           ((*pstTIF)->pFanBuf)

#define TIF_hFile              (&(*pstTIF)->hILDFX_File)

```



```

#define TIF_bWorkFileIsOpen      ((*pstTIF)->bWorkFileIsOpen)

/*-----
 * TIF_TotalRecordCount is the value returned by ILDFX_GetRecordCount().
 * This count includes both "control" records and "data" records. There are
 * N control records, numbered zero through (TIF_FIRST_ITEMNO-1), followed by
 * data records numbered TIF_FIRST_ITEMNO through (TIF_RecordCount-1).
 *-----*/
#define TIF_TotalRecordCount      ((*pstTIF)->TotalRecordCount)
#define TIF_PertinentRecordCount  ((*pstTIF)->PertinentRecordCount)
#define TIF_CurrentRecordNumber  ((*pstTIF)->CurrentRecordNumber) // virtual
#define TIF_CurrentRecordOutcome  ((*pstTIF)->CurrentRecordOutcome)
#define TIF_nSynchronize          ((*pstTIF)->nSynchronize)

#define TIF_ReconciliationOption  ((*pstTIF)->ReconciliationOption)
#define TIF_Resolution            ((*pstTIF)->Resolution)
#define TIF_AffectedItem          ((*pstTIF)->AffectedItem)

#define TIF_FileFormatVersion     ((*pstTIF)->K.FileFormatVersion)

#define TIF_AlarmFlagFieldNum     ((*pstTIF)->K.AlarmFlagFieldNum)
#define TIF_AlarmTimeFieldNum     ((*pstTIF)->K.AlarmTimeFieldNum)
#define TIF_AlarmDateFieldNum     ((*pstTIF)->K.AlarmDateFieldNum)
#define TIF_DoneFlagFieldNum      ((*pstTIF)->K.DoneFlagFieldNum)
#define TIF_StartDateFieldNum     ((*pstTIF)->K.StartDateFieldNum)
#define TIF_EndDateFieldNum       ((*pstTIF)->K.EndDateFieldNum)
#define TIF_StartTimeFieldNum     ((*pstTIF)->K.StartTimeFieldNum)
#define TIF_EndTimeFieldNum       ((*pstTIF)->K.EndTimeFieldNum)
#define TIF_ViewFieldNum          ((*pstTIF)->K.ViewFieldNum)
#define TIF_SourceIDFieldNum      ((*pstTIF)->K.SourceIDFieldNum)
#define TIF_TargetIDFieldNum      ((*pstTIF)->K.TargetIDFieldNum)
#define TIF_SourceDeltaFieldNum   ((*pstTIF)->K.SourceDeltaFieldNum)
#define TIF_TargetDeltaFieldNum   ((*pstTIF)->K.TargetDeltaFieldNum)
#define TIF_RepBasicFieldNum      ((*pstTIF)->K.RepBasicFieldNum)
#define TIF_RepExclFieldNum       ((*pstTIF)->K.RepExclFieldNum)
#define TIF_SubTypeFieldNum       ((*pstTIF)->K.SubTypeFieldNum)
#define TIF_SourceSubTypeFieldNum ((*pstTIF)->K.SourceSubTypeFieldNum)
#define TIF_KeyDateFieldNum       ((*pstTIF)->K.KeyDateFieldNum)

#define TIF_bFastSyncSourceLoad   ((*pstTIF)->K.bFastSyncSourceLoad)
#define TIF_bFastSyncTargetLoad  ((*pstTIF)->K.bFastSyncTargetLoad)
#define TIF_bSourceIsFastSyncAble ((*pstTIF)->K.bSourceIsFastSyncAble)
#define TIF_bTargetIsFastSyncAble ((*pstTIF)->K.bTargetIsFastSyncAble)

#define TIF_bSyncUsingTargetIDs   ((*pstTIF)->K.bSyncUsingTargetIDs)
#define TIF_bSyncUsingSourceIDs   ((*pstTIF)->K.bSyncUsingSourceIDs)
#define TIF_bCheckForOverlap      ((*pstTIF)->K.bCheckForOverlap)
#define TIF_bReconcileExclusions  ((*pstTIF)->K.bReconcileExclusions)
#define TIF_bReconcileRepBasic    ((*pstTIF)->K.bReconcileRepBasic)
#define TIF_bDontSyncDoneTodos    ((*pstTIF)->K.bDontSyncDoneTodos)

#define TIF_RangeOfSync           ((*pstTIF)->K.RangeOfSync)
#define TIF_RangeOfPreviousSync   ((*pstTIF)->K.RangeOfPreviousSync)
#define TIF_RangeOfSourceLoad     ((*pstTIF)->K.RangeOfSourceLoad)

#define TIF_lDateOfPreviousSync   ((*pstTIF)->K.lDateOfPreviousSync)

#define TIF_SourceEpoch          ((*pstTIF)->K.SourceEpoch)
#define TIF_TargetEpoch          ((*pstTIF)->K.TargetEpoch)

#define TIF_SourceFanoutMaxima     ((*pstTIF)->K.SourceFanoutMaxima)
#define TIF_TargetFanoutMaxima     ((*pstTIF)->K.TargetFanoutMaxima)

#define TIF_bPleaseSaveFanoutMaxima ((*pstTIF)->bPleaseSaveFanoutMaxima)
#define TIF_bNeedPriming          ((*pstTIF)->bNeedPriming)
#define TIF_bCurrentlyLoggingAndCountingRecords \
    ((*pstTIF)->bCurrentlyLoggingAndCountingRecords)
#define TIF_nWhich                ((*pstTIF)->nWhich)
#define TIF_szWorkFile            ((*pstTIF)->szWorkFile)
#define TIF_bUserCancel           ((*pstTIF)->bUserCancel)
#define TIF_bSkipLeaveAlones       ((*pstTIF)->bSkipLeaveAlones)
#define TIF_bFastSyncLoad         ((*pstTIF)->bFastSyncLoad)
#define TIF_bFastSyncUnload       ((*pstTIF)->bFastSyncUnload)
#define TIFLOG                    ((*pstTIF)->lpsILLog)
#define TIF_phase                 ((*pstTIF)->phase)

```

```

#define TIF_origin                ((*pstTIF)->origin)
#define TIF_CRPolicy              ((*pstTIF)->CRPolicy)
#define TIF_tr                    ((*pstTIF)->tr)
#define TIF_pILTIF                ((*pstTIF)->pILTIF)

#define TIF_PreviousILTRPhase     ((*pstTIF)->PreviousILTRPhase)
#define TIF_RecordHasBeenPutSinceLastGet ((*pstTIF)->RecordHasBeenPutSinceLastGet)

#define TIF_szRepStartPseudoFldName ((*pstTIF)->szRepStartPseudoFldName)
#define TIF_szRepStopPseudoFldName  ((*pstTIF)->szRepStopPseudoFldName)
#define TIF_szRepExpPseudoFldName   ((*pstTIF)->szRepExpPseudoFldName)
#define TIF_szTrue                  ((*pstTIF)->szTrue)
#define TIF_szFalse                 ((*pstTIF)->szFalse)

#define TIF_GoalPost              ((*pstTIF)->GoalPost)
#define TIF_GoalPostForNextPass   ((*pstTIF)->GoalPostForNextPass)

#define TIF_bCountingOutcomes      ((*pstTIF)->bCountingOutcomes)
#define TIF_OutcomeCounts          ((*pstTIF)->OutcomeCounts)
#define TIF_LeaveAloneCount         (TIF_OutcomeCounts.LeaveAloneCount)
#define TIF_AddCount               (TIF_OutcomeCounts.AddCount)
#define TIF_UpdateCount            (TIF_OutcomeCounts.UpdateCount)
#define TIF_ReplaceCount           (TIF_OutcomeCounts.ReplaceCount)
#define TIF_DeleteCount            (TIF_OutcomeCounts.DeleteCount)
#define TIF_IgnoreCount            (TIF_OutcomeCounts.IgnoreCount)

#define TIFTableAutomatic          ((*pstTIF)->TableAutomatic)

//----- ILDFX Record Numbers
#define TIF_RECORD_ZERO 0          // Target Field List is stored here
#define TIF_RECORD_ONE  1          // Source Field List is stored here
#define TIF_RECORD_TWO  2          // TIF_KSTRUCT is stored here
#define TIF_FIRST_ITEMNO 3         // User data is stored starting here

//----- TIF Current Record Number - special values
#define TIF_POSITION_ABOVE_TOP (TIF_FIRST_ITEMNO-1)
#define TIF_POSITION_BELOW_BOTTOM (-4)

//----- special TIF-internal fields:
#define TIF_INST_LIST_FIELD "\2InstanceArray"
#define TIF_INST_LIST_FIELDNUM 0

//----- arbitrary limit on Group Size
#define TIF_MAX_GROUP_SIZE 30000
#define TIF_MAX_SKG_SIZE TIF_MAX_GROUP_SIZE
#define TIF_MAX_FIG_SIZE 1000
#define TIF_MAX_CIG_SIZE 3          // one P-item, one T-item, one S-item
#define TIF_MAX_CONFLICTS 10

//----- offsets in ILDFX EXDATA:
#define TIF_NO_SLOT                (-1) // says NO, don't look at EXDATA!

#define TIF_NEXT_IN_CIG_SLOT 0      // Corresponding Item Group
#define TIF_NEXT_IN_SKG_SLOT 1      // Same Keyfields Group
#define TIF_NEXT_IN_FIG_SLOT 2      // Fanned Instance Group
#define TIF_FLAGS_SLOT 3
#define TIF_KEYFIELDS_HASH_SLOT 4
#define TIF_SOURCEID_HASH_SLOT 5
#define TIF_TARGETID_HASH_SLOT 6
#define TIF_NKFIELDS_HASH_SLOT 7    // Hash of Non-Key Not-No-Reconcile Flds
#define TIF_NONDATE_HASH_SLOT 8     // Hash of Non-Date Non-Key Not-No-Reconcile Fields
#define TIF_DESC_HASH_SLOT TIF_NONDATE_HASH_SLOT // for SmartMerge
#define TIF_START_DTTM_SLOT 9
#define TIF_END_DTTM_SLOT 10
#define TIF_FLAGS2_SLOT 11
#define TIF_REP_EXCL_HASH_SLOT 12

#define TIF_EXDATA_PER_RECORD 13

//----- access macros for ILDFX EXDATA values
#define TIFX(f, i, slot) ILDFX_EXDATA(f, i, slot)
#define TIFX_NEXT_IN_GROUP(f,i,linkage_slot) TIFX(f, i, linkage_slot)
#define TIFX_NEXT_IN_CIG(f,i) TIFX_NEXT_IN_GROUP(f, i, TIF_NEXT_IN_CIG_SLOT)

```

```

#define TIFX_NEXT_IN_SKG(f,i) TIFX_NEXT_IN_GROUP(f, i, TIF_NEXT_IN_SKG_SLOT)
#define TIFX_NEXT_IN_FIG(f,i) TIFX_NEXT_IN_GROUP(f, i, TIF_NEXT_IN_FIG_SLOT)

#define TIFX_FLAGS(f,i) TIFX(f, i, TIF_FLAGS_SLOT)
#define TIFX_KEYFIELDS_HASH(f,i) TIFX(f, i, TIF_KEYFIELDS_HASH_SLOT)
#define TIFX_SOURCEID_HASH(f,i) TIFX(f, i, TIF_SOURCEID_HASH_SLOT)
#define TIFX_TARGETID_HASH(f,i) TIFX(f, i, TIF_TARGETID_HASH_SLOT)
#define TIFX_NKFIELDS_HASH(f,i) TIFX(f, i, TIF_NKFIELDS_HASH_SLOT)
#define TIFX_NONDATE_HASH(f,i) TIFX(f, i, TIF_NONDATE_HASH_SLOT)
#define TIFX_DESC_HASH TIFX_NONDATE_HASH
#define TIFX_START_DTTM(f,i) TIFX(f, i, TIF_START_DTTM_SLOT)
#define TIFX_END_DTTM(f,i) TIFX(f, i, TIF_END_DTTM_SLOT)
#define TIFX_FLAGS2(f,i) TIFX(f, i, TIF_FLAGS2_SLOT)
#define TIFX_REP_EXCL_HASH(f,i) TIFX(f, i, TIF_REP_EXCL_HASH_SLOT)

#define TIFX_START_DATE(f,i) (TIFX_START_DTTM(f,i) / 1440)
#define TIFX_END_DATE(f,i) (TIFX_END_DTTM(f,i) / 1440)

#define TIFX_START_TIME(f,i) (TIFX_START_DTTM(f,i) % 1440)
#define TIFX_END_TIME(f,i) (TIFX_END_DTTM(f,i) % 1440)

//----- structure of FLAGS word in EXDATA:
#define TIF_IS_UNANALYZED 0x00000001L
#define TIFX_IS_UNANALYZED(f,i) (TIFX_FLAGS(f,i) & TIF_IS_UNANALYZED)

//----- the GARBAGE and UNANALYZED flags use the same bit!!
#define TIF_IS_GARBAGE TIF_IS_UNANALYZED

//----- values for ORIGIN part of FLAGS word
#define TIF_FROM_PREVIOUS 0x00000002L
#define TIF_FROM_TARGET 0x00000004L
#define TIF_FROM_SOURCE 0x00000006L
#define TIF_ORIGIN_MASK 0x00000006L // does NOT include 'unanalyzed' bit

#define TIF_IS_FANNED_FOR_TARGET 0x00000008L

#define TIF_OUTCOME_MASK 0x00000FF0L
/*-----
 * We use the TIF OUTCOME bits differently, depending on whether
 * we're doing SmartMerge or Synchronization. For SmartMerge
 * the following outcome bit definitions are used:
 *-----*/
#define TIF_OUTCOME_ADD_ITEM_TO_TARGET 0x00000010L
#define TIF_OUTCOME_REPLACE_ITEM_IN_TARGET 0x00000020L
#define TIF_OUTCOME_UPDATE_ITEM_IN_TARGET 0x00000040L

#define TIF_ITEM_WILL_END_UP_IN_TARGET_MASK 0x00000070L // add, rep, or upd
#define TIFX_OUTCOME(f,i) (TIFX_FLAGS(f,i) & TIF_OUTCOME_MASK)

#define TIFX_ITEM_WILL_END_UP_IN_TARGET(f,i) \
    (TIFX_FLAGS(f,i) & TIF_ITEM_WILL_END_UP_IN_TARGET_MASK)

#define TIFX_ITEM_WILL_BE_ADDED_TO_TARGET(f,i) \
    (TIFX_FLAGS(f,i) & TIF_OUTCOME_ADD_ITEM_TO_TARGET)

#define TIFX_ITEM_WILL_NOT_END_UP_IN_TARGET(f,i) \
    (!TIFX_ITEM_WILL_END_UP_IN_TARGET(f,i))

#define TIF_OUTCOME_IGNORE_DUPLICATE_ITEM 0x00000080L
#define TIF_OUTCOME_IGNORE_CONFLICTING_ITEM 0x00000100L
#define TIF_OUTCOME_IGNORED_MASK 0x00000180L
#define TIFX_ITEM_IS_IGNORED(f,i) \
    ((TIFX_FLAGS(f,i) & TIF_OUTCOME_IGNORED_MASK) != 0)

#define TIF_OUTCOME_DELETE_ITEM_FROM_TARGET 0x00000200L

//----- OBSOLETE means that item is to be replaced or updated by another
#define TIF_OUTCOME_OBSOLETE_ITEM 0x00000400L
#define TIF_OUTCOME_IGNORED_OR_OBSOLETE 0x00000580L

#define TIF_ITEM_OBSOLETE_ANOTHER 0x00000800L
#define TIFX_ITEM_OBSOLETE_ANOTHER(f,i) \
    (TIFX_FLAGS(f,i) & TIF_ITEM_OBSOLETE_ANOTHER)

#define TIFX_ITEM_IS_OBSOLETE(f,i) \

```

```

        (TIFX_FLAGS(f,i) & TIF_OUTCOME_OBSOLETE_ITEM)
#define TIFX_ITEM_ISNT_OBSOLETE(f,i) (!TIFX_ITEM_IS_OBSOLETE(f,i))

/*-----
 * For Synchronization of "peer" items, the following outcome bit
 * definitions are used:
 *-----*/
#define TIFSYNC_TARGET_WINS          1L
#define TIFSYNC_SOURCE_WINS          2L
#define TIFSYNC_IGNORE               3L
#define TIF_SYNC_OUTCOME_COUNT       3L

#define TIF_OUTCOME_SHIFT             4 // bit positions away from 1's
#define TIF_OUTCOME_MUL               0x10L
#define TIF_OUTCOME_SYNC_TARGET_WINS (TIFSYNC_TARGET_WINS * TIF_OUTCOME_MUL)
#define TIF_OUTCOME_SYNC_SOURCE_WINS (TIFSYNC_SOURCE_WINS * TIF_OUTCOME_MUL)
#define TIF_OUTCOME_SYNC_IGNORE      (TIFSYNC_IGNORE * TIF_OUTCOME_MUL)

#define TIF_OUTCOME_SYNC_MASK         0x30L

/*-----
 * NOTE: for synchronization these 5 bits are UNUSED: 0xF40
 *-----*/

//----- flags used for ID-based matches
#define TIF_ID_MATCH          0x0000C000L
#define TIF_ID_MATCH_PT      0x00004000L // Previous & Target
#define TIF_ID_MATCH_PS      0x00008000L // Previous & Source

//----- flags used for KF-based matches
#define TIF_KFM_EXACT1        0x00001000L
#define TIF_KFM_EXACT2        0x00002000L
#define TIF_KFM_EXACT12       0x00003000L

#define TIF_KFM_INEXACT1      0x00010000L
#define TIF_KFM_INEXACT2      0x00020000L
#define TIF_KFM_INEXACT12     0x00030000L

#define TIF_KFM_STRONG1       0x00011000L
#define TIF_KFM_STRONG2       0x00022000L
#define TIF_KFM_STRONG12      0x00033000L

#define TIF_KFM_ANY1          0x00011000L
#define TIF_KFM_ANY2          0x00022000L
#define TIF_KFM_ANY12         0x00033000L

//----- flags used when we match fanned instances to Master Recurrence Items
#define TIF_IS_SYNTHETIC_MASTER 0x00040000L // flag set when we create a pseudo-master
#define TIFX_IS_SYNTHETIC_MASTER(f,i) \
    (TIFX_FLAGS(f,i) & TIF_IS_SYNTHETIC_MASTER)

#define TIF_IS_GOBLED_UP_INSTANCE 0x00080000L // flag set when we gobble up an instance
#define TIFX_IS_GOBLED_UP_INSTANCE(f,i) \
    (TIFX_FLAGS(f,i) & TIF_IS_GOBLED_UP_INSTANCE)
#define TIFX_ISNT_GOBLED_UP_INSTANCE(f,i) (!TIFX_IS_GOBLED_UP_INSTANCE(f,i))

/*-----
 * CIG Type identifiers:
 *
 * 3-digit Source, Previous, Target string:
 *
 * a ZERO digit means item absent in this domain
 * a ONE digit means item has base value for this domain
 * a TWO digit means item has different value in this domain
 * a THREE digit means item has yet another value in this domain
 *-----*/
#define TIF_CIG_TYPE_SHIFT 20 // bit-positions away from 1s place
#define TIFCIG_MUL         0x00100000L
#define TIF_CIG_TYPE_MASK 0x01F00000L

//-----values used for NOT-PREVIOUSLY-SYNCHRONIZED items
#define TIFCIG_000 0 // no cig type assigned (yet)
#define TIFCIG_001 1 // item is present in target only ("new in target")
#define TIFCIG_100 2 // item is present in source only ("new in source")
#define TIFCIG_101 3 // item is identical in Source and Target

```

```

#define TIFCIG_102 4 // new source item <> new target item

//-----values used for PREVIOUSLY SYNCHRONIZED items
#define TIFCIG_111 5 // item is unchanged across the board
#define TIFCIG_112 6 // item CHANGED in Target since last sync
#define TIFCIG_110 7 // item DELETED from Target since last sync
#define TIFCIG_211 8 // item CHANGED in Source since last sync
#define TIFCIG_212 9 // item CHANGED IDENTICALLY in Src & Target
#define TIFCIG_213 10 // item CHANGED DIFFERENTLY in Src & Target
#define TIFCIG_210 11 // item CHANGED in Source, DELETED from Target
#define TIFCIG_011 12 // item DELETED from Source since last sync
#define TIFCIG_012 13 // item DELETED from Source, CHANGED in Target
#define TIFCIG_010 14 // item DELETED from both Source & Target

#define TIFCIG_132 15 // 102 conflict resolved interactively
                        // to a "compromise" value; always UpdateBoth.
#define TIFCIG_13F 16 // Used when UpdateBoth is Fanned to Target
#define TIFCIG_MAX 16

#define TIF_CIG_TYPE_000 (TIFCIG_000 * TIFCIG_MUL) // 0x00000000L
#define TIF_CIG_TYPE_001 (TIFCIG_001 * TIFCIG_MUL) // 0x00100000L
#define TIF_CIG_TYPE_100 (TIFCIG_100 * TIFCIG_MUL) // 0x00200000L
#define TIF_CIG_TYPE_101 (TIFCIG_101 * TIFCIG_MUL) // 0x00300000L
#define TIF_CIG_TYPE_102 (TIFCIG_102 * TIFCIG_MUL) // 0x00400000L
#define TIF_CIG_TYPE_111 (TIFCIG_111 * TIFCIG_MUL) // 0x00500000L
#define TIF_CIG_TYPE_112 (TIFCIG_112 * TIFCIG_MUL) // 0x00600000L
#define TIF_CIG_TYPE_110 (TIFCIG_110 * TIFCIG_MUL) // 0x00700000L
#define TIF_CIG_TYPE_211 (TIFCIG_211 * TIFCIG_MUL) // 0x00800000L
#define TIF_CIG_TYPE_212 (TIFCIG_212 * TIFCIG_MUL) // 0x00900000L
#define TIF_CIG_TYPE_213 (TIFCIG_213 * TIFCIG_MUL) // 0x00a00000L
#define TIF_CIG_TYPE_210 (TIFCIG_210 * TIFCIG_MUL) // 0x00b00000L
#define TIF_CIG_TYPE_011 (TIFCIG_011 * TIFCIG_MUL) // 0x00c00000L
#define TIF_CIG_TYPE_012 (TIFCIG_012 * TIFCIG_MUL) // 0x00d00000L
#define TIF_CIG_TYPE_010 (TIFCIG_010 * TIFCIG_MUL) // 0x00e00000L
#define TIF_CIG_TYPE_132 (TIFCIG_132 * TIFCIG_MUL) // 0x00f00000L
#define TIF_CIG_TYPE_13F (TIFCIG_13F * TIFCIG_MUL) // 0x01000000L

/*-----
 * The 'Mex CIG' bit is set when the original CIG type and SyncOutcome bits
 * are altered to encode instructions for an ExclusionList Merge. When this
 * is done the original CIG type is stored in FLAGS2.
 *-----*/
#define TIF_MEX_CIG 0x02000000L // cigtype altered by MEX

/*-----
 * The 'Mex Only' bit is used in conjunction with CIG types 213, 132, and
 * 13F (132 after Fanning to Target), to indicate that the only field
 * values to be updated are the Exclusion List and the Exclusion Count
 * in the _repBasic field. See tifmex.cpp and tif.cpp\GetFieldByIndex.
 *-----*/
#define TIF_MEX_ONLY 0x04000000L // only update Exclusion List & Count

//----- when FastSync loads an item into TIF with _Delta=D, the ZOMBIE bit is set
#define TIF_ZOMBIE 0x08000000L // item is a zombie

/*-----
 * Assume that the endpoint origins are X and Y, where X is either Source
 * or Target, and Y is either Target or Source.
 *
 * The 'WRONG_SST' bit, when turned ON for an item whose origin is X,
 * says that the item must NOT be unloaded by Y due to SubType Filtering.
 *-----*/
#define TIF_WRONG_SST 0x10000000L // wrong Section SubType for xfer
#define TIFX_WRONG_SST(f,i) ((TIFX_FLAGS(f,i) & TIF_WRONG_SST) != 0)
#define TIFX_RIGHT_SST(f,i) ((TIFX_FLAGS(f,i) & TIF_WRONG_SST) == 0)

#define TIF_BYSTANDER 0x20000000L

//----- the IGNORE bitmask excludes records that most people don't want to see
#define TIF_IGNORE (TIF_ZOMBIE | TIF_IS_GARBAGE | TIF_BYSTANDER)

#define TIF_ITEM_IS_RECURRING 0x40000000L

#define TIFX_ITEM_IS_RECURRING(f,i) \
    ((TIFX_FLAGS(f,i) & TIF_ITEM_IS_RECURRING) != 0)

```

```

#define TIFX_ITEM_ISNT_RECURRING(f,i) \
    ((TIFX_FLAGS(f,i) & TIF_ITEM_IS_RECURRING) == 0)

/*-----
 * when ILTIFFanItem is called we mark BOTH master & instances as
 * Fanned For Target or Fanned For Source.
 *-----*/
#define TIF_IS_FANNED_FOR_SOURCE 0x80000000L

#define TIF_IS_FANNED_FOR_WHOM \
    (TIF_IS_FANNED_FOR_SOURCE | TIF_IS_FANNED_FOR_TARGET)

#define TIFX_ORIGIN(f,i) (TIFX_FLAGS(f,i) & TIF_ORIGIN_MASK)
#define TIFX_CIG_TYPE(f,i) (TIFX_FLAGS(f,i) & TIF_CIG_TYPE_MASK)

/*-----
 * Flag bits in the FLAGS2 doubleword:
 *
 * TIF2_IS_PCLONE (for FastSync only) says item was generated when
 * TIFSyncDigestFastLoad called TweakFastSyncCIG which in turn
 * called AddPCloneToCIG. (As opposed to being loaded into TIF
 * by virtue of a translator calling ILTIFFPutRecord.)
 *-----*/
#define TIF2_IS_A_PCLONE 0x00000001L

/*-----
 * next 2 flags are used when All Fields Hash is tweaked to force re-fanning
 *-----*/
#define TIF2_AFH_BUMPED_UP 0x00000002L
#define TIF2_AFH_BUMPED_DOWN 0x00000004L

/*-----
 * next 2 flags are used, for sync, when unloader calls ILTIFFRejectOutcome
 *-----*/
#define TIF2_SOURCE_REJECT 0x00000008L
#define TIF2_TARGET_REJECT 0x00000010L

//----- next flag is set when item is a Completed TODO
#define TIF2_DONE_TODO 0x00000020L
#define TIFX_IS_DONE_TODO(f,i) ((TIFX_FLAGS2(f,i) & TIF2_DONE_TODO) != 0)

//----- next flag is set when an appt or TODO item is outside the Date Range
#define TIF2_OUT_OF_RANGE 0x00000040L
#define TIFX_IS_OUT_OF_RANGE(f,i) ((TIFX_FLAGS2(f,i) & TIF2_OUT_OF_RANGE) != 0)

//----- next 2 flag bits are used to remember origins of ADD-ACROSS products
#define TIF2_WAS_CIGTYPE_213 0x00000080L
#define TIF2_WAS_CIGTYPE_102 0x00000100L

//----- next 4 flag bits are used to keep track of list traversal progress
#define TIF2_MARK_DUMP1 0x00000200L
#define TIF2_MARK_DUMP2 0x00000400L
#define TIF2_MARK_SKG1 0x00000800L
#define TIF2_MARK_SKG2 0x00001000L

//----- special DELETE_ME bits set by ILTIFFanItem (sync only)
#define TIF2_DELETE_SOURCE 0x00002000L
#define TIF2_DELETE_TARGET 0x00004000L

//----- next flag is set when a cig type 101 or 102 that is entirely out
//----- of range is chopped into 2 pieces (a 100 and a 001) by set_cig_type
#define TIF2_WAS_OUTRANGE_10X 0x00008000L

//----- next flag is used to avoid redundant master/instance matching
#define TIF2_FOUND_INSTANCES 0x00010000L

//----- next 3 bits are unused
#define TIF2_UNUSED_A 0x000E0000L

//----- next 5 bits are used to store the original CIG type when the need to
//----- Merge Exclusion Lists forces a switch to a different (MEX) cig type.
#define TIF2_ORIGINAL_CIG_TYPE_MASK 0x01F00000L // must match TIF_CIG_TYPE_MASK

//----- next 7 bits are unused
#define TIF2_UNUSED_B 0xFE000000L

```



```

//----- offset in spt[] array passed to TIFTablePickRecordsForSync() function
//----- SPT stands for Source, Previous, Target
#define TIF_SPT_S 0
#define TIF_SPT_P 1
#define TIF_SPT_T 2

#define TIF_CRPolicy_Remove_SS_Duplicates \
    ((TIF_CRPolicy & ILXTR_CR_POLICY_REMOVE_SS_DUPLICATES) != 0)

#define TIF_CRPolicy_Detect_SS_Conflicts \
    ((TIF_CRPolicy & ILXTR_CR_POLICY_DETECT_SS_CONFLICTS) != 0)

#define TIF_FieldCount(pp) ((*pp)->pFieldList->ActualFieldCount)
#define TIF_FieldCount2(pfl) ((pfl)->ActualFieldCount)

#define TIF_FieldDesc(pp, fldnum) ((*pp)->pFieldList->pFieldDescs[fldnum])
#define TIF_FieldDesc2(pfl, fldnum) ((pfl)->pFieldDescs[fldnum])

#define TIF_FieldDefaultValue(pp, fldnum) (TIF_FieldDesc(pp, fldnum).szDefault)
#define TIF_FieldDefaultValue2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).szDefault)

#define TIF_FieldToGetDefaultValueFrom(pp, fldnum) \
    (TIF_FieldDesc(pp, fldnum).FieldToGetDefaultValueFrom)

#define TIF_FieldFormat(pp, fldnum) (TIF_FieldDesc(pp, fldnum).szFormat)
#define TIF_FieldFormat2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).szFormat)

#define TIF_FieldType(pp, fldnum) (TIF_FieldDesc(pp, fldnum).FieldType)
#define TIF_FieldType2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).FieldType)

#define TIF_FieldMaxLength(pp, fldnum) (TIF_FieldDesc(pp, fldnum).MaxLength)
#define TIF_FieldMaxLength2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).MaxLength)

#define TIF_FieldMaxMappedLength(pp, fldnum) \
    (TIF_FieldDesc(pp, fldnum).MaxMappedLength)

#define TIF_FieldName(pp, fldnum) (TIF_FieldDesc(pp, fldnum).szFldName)
#define TIF_FieldName2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).szFldName)

#define TIF_FieldNum(pp, fldnum) (TIF_FieldDesc(pp, fldnum).FieldNum)
#define TIF_FieldNum2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).FieldNum)

#define TIF_RelatedFieldName(pp, fldnum) \
    (TIF_FieldDesc(pp, fldnum).RelatedFieldName)

#define TIF_RelatedFieldName2(pfl, fldnum) \
    (TIF_FieldDesc2(pfl, fldnum).RelatedFieldName)

#define TIF_RelatedFieldNum(pp, fldnum) \
    (TIF_FieldDesc(pp, fldnum).RelatedFieldNum)

#define TIF_FieldIsAdded(pp, fldnum) (TIF_FieldDesc(pp, fldnum).FieldIsAdded)
#define TIF_FieldIsAdded2(pfl, fldnum) (TIF_FieldDesc2(pfl, fldnum).FieldIsAdded)

#define TIF_isPriorityField(pp, fldnum) \
    ((TIF_FieldAttributes(pp, fldnum) & ILTB_ATT_PRIORITY) != 0)

#define TIF_FieldAttributes(pp, fldnum) \
    (TIF_FieldDesc(pp, fldnum).FieldAttributes)

#define TIF_FieldAttributes2(pfl, fldnum) \
    (TIF_FieldDesc2(pfl, fldnum).FieldAttributes)

#define TIF_ExtraAttributes(pp, fldnum) \
    (TIF_FieldDesc(pp, fldnum).ExtraAttributes)

#define TIF_ExtraAttributes2(pfl, fldnum) \
    (TIF_FieldDesc2(pfl, fldnum).ExtraAttributes)

#define TIF_isKeyField(pp, fldnum) \
    ((TIF_FieldAttributes(pp, fldnum) & ILTB_ATT_KEY_FIELD) != 0)

#define TIF_isDescriptionField(pp, fldnum) \
    ((TIF_FieldAttributes(pp, fldnum) & ILTB_ATT_VIEW) != 0)

```

```

#define TIF_isntKeyField(pp, fldnum) (!TIF_isKeyField(pp, fldnum))

#define TIF_isAutoFillInField(pp, fldnum) \
    ((TIF_ExtraAttributes(pp, fldnum) & TIFEA_AUTO_FILLIN) != 0)

#define TIF_FieldIsMapped(pp, fldnum) \
    ((TIF_ExtraAttributes(pp, fldnum) & TIFEA_ISNT_MAPPED) == 0)

#define TIF_FieldIsntMapped(pp, fldnum) \
    ((TIF_ExtraAttributes(pp, fldnum) & TIFEA_ISNT_MAPPED) != 0)

#define TIF_FieldFirstLineMapped(pp, fldnum) \
    ((TIF_ExtraAttributes(pp, fldnum) & TIFEA_FIRST_LINE_MAPPED) != 0)

#define TIF_isRepBasicField(pp, fldnum) (fldnum == (*pp)->K.RepBasicFieldNum)

#define TIF_isRepExclField(pp, fldnum) (fldnum == (*pp)->K.RepExclFieldNum)

#define TIF_isntMatchField(pp, fldnum) \
    ((TIF_FieldAttributes(pp, fldnum) & ILTB_ATT_NO_RECONCILE) != 0)

#define TIF_isMatchField(pp, fldnum) (!TIF_isntMatchField(pp, fldnum))

#define TIF_isMultiLineField(pp, fldnum) \
    ((TIF_FieldAttributes(pp, fldnum) & ILTB_ATT_MULTILINE) != 0)

#define TIF_isSourceIDField(pp, fldnum) (fldnum == (*pp)->K.SourceIDFieldNum)

#define TIF_isTargetIDField(pp, fldnum) (fldnum == (*pp)->K.TargetIDFieldNum)

#define TIF_isSourceDeltaField(pp, fldnum) \
    (fldnum == (*pp)->K.SourceDeltaFieldNum)

#define TIF_isTargetDeltaField(pp, fldnum) \
    (fldnum == (*pp)->K.TargetDeltaFieldNum)

//----- Field Offsets are all set relative to BYTE ZERO of the record structure
#define TIF_FirstOffset(pp) ((*pp)->pFieldList->FirstOffset)
#define TIF_FirstOffset2(pfl) ((pfl)->FirstOffset)
#define TIF_FieldOffset(pr, fldnum) ((pr)->pFieldValues[fldnum].FieldOffset)
#define TIF_FieldLength(pr, fldnum) ((pr)->pFieldValues[fldnum].lFldSize)

inline IL_PSTR TIF_FieldData
( TIF_RECORD_VALUE_PTR pr,
  INT16 fldnum )
{
    INT32 offset = TIF_FieldOffset(pr, fldnum);
    IL_PSTR pData = (IL_PSTR) pr;
    return (pData + offset);
}

//----- when doing Master/Instance correlations, don't fan forever...
#define TIFSYNC_MAX_FAN_COUNT 500

typedef struct
{
    long lDate;
    INT32 lSlaveItem;
    char bExcludedFromMaster;
    char cWhichFanout;          //----- use #defines below
}
TIFSYNC_INSTANCE_TYPE;

//----- values for cWhichFanout
#define TIFSYNC_CURR_FANOUT 1
#define TIFSYNC_PREV_FANOUT 2

//----- special HASH values that have special meanings
#define TIFHASH_NONE 0 // means nothing has been hashed
#define TIFHASH_SPECIAL 1 // "this is a Fig Master; ID-bearing FIG attached"
#define TIFHASH_FIRST_NORMAL 2 // lowest un-distinguished hash value

//----- special pseudo-fieldnumbers used when calling TIFHash

```



```

#define TIFHASH_STARTSTOP          (-99)      // Start&Stop Recurrence Dates

/*-----
 * definitions for TIF logging from ILDFX handle
 *-----*/
#define TIFXlogsz(s)                ILDFXlogsz      (phFile,s)
#define TIFXlogszsz(s,t)            ILDFXlogszsz    (phFile,s,t)
#define TIFXlogsz3(s,t,u)           ILDFXlogsz3     (phFile,s,t,u)
#define TIFXlogsz3ul(s,t,u,v)       ILDFXlogsz3ul    (phFile,s,t,u,v)
#define TIFXlogszul(s,t)            ILDFXlogszul     (phFile,s,t)
#define TIFXlogszszul(s,t,u)        ILDFXlogszszul   (phFile,s,t,u)
#define TIFXlogszulul(s,t,u)        ILDFXlogszulul   (phFile,s,t,u)
#define TIFXlogsz2ul2(s,t,u,v)      ILDFXlogsz2ul2   (phFile,s,t,u,v)
#define TIFXlogszul3(s,t,u,v)       ILDFXlogszul3    (phFile,s,t,u,v)

/*-----
 * definitions for logging in low-level TIF environment
 *-----*/
#define TIFlogsz(s)                 ILLOG_logsz      (TIFLOG,ILLOG_ALWAYS,s)
#define TIFlogszsz(s,t)             ILLOG_logszsz    (TIFLOG,ILLOG_ALWAYS,s,t)
#define TIFlogsz3(s,t,u)            ILLOG_logsz3     (TIFLOG,ILLOG_ALWAYS,s,t,u)
#define TIFlogsz3ul(s,t,u,v)        ILLOG_logsz3ul   (TIFLOG,ILLOG_ALWAYS,s,t,u,v)
#define TIFlogszul(s,t)             ILLOG_logszul     (TIFLOG,ILLOG_ALWAYS,s,t)
#define TIFlogszszul(s,t,u)         ILLOG_logszszul   (TIFLOG,ILLOG_ALWAYS,s,t,u)
#define TIFlogszulul(s,t,u)         ILLOG_logszulul   (TIFLOG,ILLOG_ALWAYS,s,t,u)
#define TIFlogsz2ul2(s,t,u,v)       ILLOG_logsz2ul2   (TIFLOG,ILLOG_ALWAYS,s,t,u,v)
#define TIFlogszul3(s,t,u,v)        ILLOG_logszul3    (TIFLOG,ILLOG_ALWAYS,s,t,u,v)

/*-----
 * definitions for verbosity-sensitive logging in low-level TIF environment
 *-----*/
#define TIFlog0(v,s)                ILLOG_logsz      (TIFLOG, v, s)
#define TIFlog1str(v,s,a)           ILLOG_logszsz    (TIFLOG, v, s, a)
#define TIFlog2str(v,s,a,b)         ILLOG_logsz3     (TIFLOG, v, s, a, b)
#define TIFlog1int(v,s,a)           ILLOG_logszul     (TIFLOG, v, s, (UINT32) a)
#define TIFlog2ints(v,s,a,b)        ILLOG_logszulul   ( TIFLOG, v, s, (UINT32) a, \
                                                         (UINT32) b )
#define TIFlog3ints(v,s,a,b,c)      ILLOG_logszul3    ( TIFLOG, v, s, (UINT32) a, \
                                                         (UINT32) b, (UINT32) c )

#define TIFlog1l(v,s,a,b)           ILLOG_logszszul   (TIFLOG, v, s, a, (UINT32) b)
#define TIFlog12(v,s,a,b,c)         ILLOG_logsz2ul2   ( TIFLOG, v, s, a, \
                                                         (UINT32) b, (UINT32) c )
#define TIFlog2l(v,s,a,b,c)         ILLOG_logsz3ul    (TIFLOG, v, s, a, b, (UINT32) c)

#endif //---- #ifndef _TIF_INCLUDED

```

```

#ifndef _TIFFN_INCLUDED
#define _TIFFN_INCLUDED

/*-----
 * Name:      TIFFN.H
 * Purpose:   Function prototypes for private 'TIF' functions.
 *
 * These functions should NOT be called by anyone outside of the ILTIF subsystem.
 *
 * Note that these are C++ functions, not C-callable.
 *
 * Authors:   Ken Dobson, David Boothby,
 * Copyright (c) IntelliLink Corporation, 1994-1995
 *-----*/

/*-----
 * Name:      TIFHash
 * Purpose:   function used to compute CHECKSUM of a blob of data.
 * NOTE:      same as ILDF\ILDFEX.C\ILDFKeyValue and analogous to
 *            ILDF\CHKSUM.C\ILDFCheckSum.
 * Author:    Mike Blanchette, Copyright (c) IntelliLink, 1993
 * 2nd Author: David Boothby, Copyright (c) IntelliLink, 1995
 *
 * to allow checksum calculation of a set of input blobs, rather than just
 * one blob, new first arg added to calling sequence. For first or only
 * blob, pass ZERO for 1st arg. For subsequent blobs, pass hash of blobs
 * checksummed so IL_DIST.
 *
 * data may be binary or null-terminated text. For binary supply exact length.
 * For text, it doesn't matter whether the length you supply includes or
 * excludes the null terminator -- the result will be the same.
 *
 * For text both leading & trailing whitespace is stripped
 * before computing hash.
 *
 * 3/12/95: never return value ZERO.
 *-----*/
INT32 TIFHash ( PSTTIF_TYPE pstTIF, int fieldnum, INT32 checksum,
               IL_USHPSTR pData, INT32 lData, IL_PSTR szPurpose );

/*-----
 * TIFSimpleHash -- call TIFHash for text for single-blob calculation
 *-----*/
INT32 TIFSimpleHash ( PSTTIF_TYPE pstTIF, IL_PSTR sz );

/*-----
 * TIFNormalizeRepeatDates -- compute normalized start & stop dates that
 * fit the repeat pattern. For example a yearly Christmas party starting
 * Jan 1, 1996 will have start date normalized to 12/25/96.
 *-----*/
int TIFNormalizeRepeatDates ( ILTR_PREPEAT pRepeatBlock,
                             long IL_DIST *pStartDate,
                             long IL_DIST *pStopDate );

/*-----
 * Name:      TIFComputeSearchKeyValues
 * Purpose:   Compute Hash Values and Start&End DTTM Values and
 *            store them in the exdata array
 *-----*/
int TIFComputeSearchKeyValues ( PSTTIF_TYPE pstTIF,
                               TIF_RECORD_VALUE_PTR pRecord,
                               INT32 *exdata );

int TIFSanitizeRepBasic ( PSTTIF_TYPE pstTIF,
                        ILTR_PREPEAT pRepeat,
                        ILTR_PREPEAT pRepeat2 );

int TIFSetSimpleApptSpan ( PSTTIF_TYPE pstTIF,
                          TIF_RECORD_VALUE_PTR pRecord,
                          INT32 IL_DIST *plStart,
                          INT32 IL_DIST *plEnd );

/*-----
 * CAAR functions: Conflict Analysis And Resolution
 *-----*/

```

```

int TIFSynchronization_CAAR (ILTR_PTRANS� tr);

int TIFSmartMerge_CAAR (ILTR_PTRANS� tr);

/*-----
 * Name:      TIFAdjustRecordIfChanged
 * Called by: GetResolution function, in TIFMERGE.CPP
 * Purpose:   To make storage and search key adjustments for a Reconciled
 *            Record, which the user may have edited.
 *-----*/
int TIFAdjustRecordIfChanged(PSTTIF_TYPE pstTIF, INT32 SourceItem);

/*-----
 * Name:      TIFAskUserToResolveConflict
 * Purpose:   Ask User to resolve conflict
 *            between 1 source item and N target items.
 *-----*/
int TIFAskUserToResolveConflict
( ILTR_PTRANS� tr,
  PSTTIF_TYPE pstTIF,
  INT32 SourceItem,
  INT32 NumberOfConflicts,
  INT32 IL_DIST *ConflictingTargetItems,          // IN (ARRAY)
  TIF_RECONCILIATION_OPTION IL_DIST *pResolution, // OUT
  INT32 IL_DIST *pAffectedTargetItem );          // OUT

/*-----
 * TIFGroup_GetNext -- get next item in group, where group is a linked list
 *                    of ILDFX entries, linked by the EXDATA value found
 *                    at offset 'linkSlot'. This function does some sanity
 *                    checking and returns an error code (negative number)
 *                    if the next item is insane.
 *-----*/
INT32 TIFGroup_GetNext (ILDFX_PHNDL phFile,
                       INT32 lRecordNumber,
                       int linkSlot );

#define TIFGetNextInCIG(f,n) TIFGroup_GetNext(f,n,TIF_NEXT_IN_CIG_SLOT)
#define TIFGetNextInSKG(f,n) TIFGroup_GetNext(f,n,TIF_NEXT_IN_SKG_SLOT)

/*-----
 * Name:      TIFremoveFromSKG
 *            ride around the circular list to find precursor of
 *            item to be removed, then link from previous to subsequent:
 *
 *            before: previous --> cigItem --> subsequent
 *            after:  previous --> subsequent
 *
 *            pass NULL for 3rd arg unless you need to get back the index
 *            of the predecessor of the deleted item.
 *-----*/
int TIFremoveFromSKG (ILDFX_PHNDL phFile, INT32 Item,
                     INT32 IL_DIST *pPredecessor);

//----- sanity check item
int TIFGroup_ValidateItem (ILDFX_PHNDL phFile,
                          INT32 lRecordNumber);

/*-----
 * Use one of the following TIFWHICH values when calling TIFVerifyFieldsMatch
 *-----*/
typedef enum
{
  TIFW_ALL_FIELDS,          // compare all mapped Not-No-Reconcile Fields
  TIFW_KEY_FIELDS,          // compare all mapped Key Fields
  TIFW_NONKEY_FIELDS        // compare all mapped Not-No-Reconcile Non-Key Fields
}
TIFWHICH;

/*-----
 * Name:      TIFVerifyFieldsMatch
 *
 * Compare full field values to verify exact match. This is an expensive
 * operation, done only when relevant hash values are equal.
 *-----*/

```

```

*
* RETURNS:  SUCCESS (match) or TIF_NO_MATCH or abnormal error code
*
* NOTE:  Pass FirstKey == -1 to compare the record that is currently in
*        memory as "*pRec1" with the record identified by SecondKey.
*
* WARNING:  uses the TIF_CurrentField and TIF_OriginalField buffers
*-----*/
int TIFVerifyFieldsMatch ( PSTTIF_TYPE pstTIF,
                          TIF_RECORD_VALUE_PTR pRec1,
                          INT32 FirstKey,
                          INT32 SecondKey,
                          TIFWHICH nWhichFields );

/*-----
* Name:      TIFRecordAddFieldValue
*
* Add field value, either to Current Record Value structure, or to ILTIF
* Cache, depending on circumstances.
*
* You can either pass a Field Name, in 2nd arg, or pass a Field Number,
* in 3rd arg.  When passing a Field Number,
* put a zero-length string (not NULL pointer) in the 2nd arg.
*
* Pass bMapCharsIfNonBinary=TRUE if you want text-type fields to be
* subject to character mapping from endpoint-specific encoding to
* IntelliLink intermediate encoding of line terminators and other chars.
*
* NOTE:  for non-binary fields, the 'lData' arg is may be ignored ... we use
*        IL_STRLEN to compute the length of non-binary 'pData'.  But if
*        'lData' has the TIF_LENGTH_GUARANTEED bit set then we respect lData.
*-----*/
#define TIF_LENGTH_GUARANTEED 0x40000000L

int TIFRecordAddFieldValue
( PSTTIF_TYPE pstTIF,
  TIF_FIELDLIST_PTR pFieldList, // either Target or Source Field List
  IL_PSTR szFldName,           // supply either a field name...
  INT16 fieldnum,              // ...or a field number
  IL_PSTR pData,               // Field Data to be added
  LONG lData,                  // Length of the data
  BOOLEAN bMapCharsIfNonBinary, // for character mapping of text fields
  ILUT_PBUFFER pRecord );      // Buffer for record

// Initialize the underlying TIF mechanism
int TIFInit
( PSTTIF_TYPE pstTIF,
  ILTR_PTRANSL tr,
  int nNumOfFlds );

int TIFLoadKStruct (ILDFX_PHNDL phFile, TIF_KSTRUCT *pK);

/*-----
* TIFStartNextPhase
*
* Signals beginning of a new phase, and indicates end of previous phase
*
* use TIF_PHASE_XXX definitions from ILTIF.H
*-----*/
int TIFStartNextPhase (ILTR_PTRANSL tr, INT16 phase);

int TIFSaveFanoutMaxima (ILTR_PTRANSL tr);

// Close the underlying TIF mechanism
int TIFTerminate (PSTTIF_TYPE pstTIF, ILTR_PTRANSL tr);

// Free buffers that hang off the TIF structure
void TIFFreeBuffers (ILTR_PTRANSL tr, PSTTIF_TYPE pstTIF);

/*-----
* Name:      TIFInitRecord
* Purpose:  Initialize the TIF structure representing an ILDFX record buffer
*-----*/
int TIFInitRecord ( PSTTIF_TYPE          pstTIF,
                   TIF_FIELDLIST_PTR    pFL,

```

```

        ILUT_PBUFFER          pRecord );

// Returns how many fields are currently defined in TIF
int TIFHowManyField
( PSTTIF_TYPE pstTIF,
  INT32 *pFieldCount );          // Number of fields defined

// Returns field name given an index
int TIFGetFieldName
( PSTTIF_TYPE pstTIF,
  INT16 fieldnumber,             // zero-based index of fieldname
  IL_PSTR szFldName );          // buffer for fieldname

// Describe one field in the field list
int TIFDefineOneField
( PSTTIF_TYPE pstTIF,
  TIF_FIELDLIST_PTR pFieldList, // Field List (primary or SOURCE cache)
  IL_PSTR szFldName,            // Field name
  LONG lFldSize,               // Field size
  int nFldType,                // Field type
  IL_PSTR szFormat,            // Field format
  IL_PSTR szRelFldName,        // Associated field
  ILTB_ATTRIB tFlags,          // Field Attributes
  INT32 *pExtraAttributes,     // array of extra field attributes
  BOOLEAN bPositive,           // Is Assoc field added
  IL_PSTR szDefault,           // Default data for field
  ILTR_NDX iltrFieldnum );     // Field index in ILTR Field List

// Locate the definition of a field that TIF knows about
int TIFLookupFieldDefinition
( PSTTIF_TYPE pstTIF,
  IL_PSTR szFldName,            // IN: Field name
  TIF_FIELD_DESC_PTR IL_DIST *ppDefinition ); // OUT: ptr to Definition

/*-----
 * Name:      TIFPutFieldByIndex -- put field w/o doing any character mapping
 *-----*/
int TIFPutFieldByIndex
( PSTTIF_TYPE pstTIF,
  INT16 fieldnum,
  IL_PANY pFldData,
  INT32 len );                // len is only relevant for BINARY fields

// Make sure that all fields that are supposed to have a value
// are filled in with either a user supplied default, or if there
// is no user supplied default then a TIF default value based on
// the type
int TIFFillDefaults
( PSTTIF_TYPE pstTIF,
  ILTR_PTRANSL tr,             // tr struct
  ILUT_PBUFFER pRecord );     // Pointer to record buffer

/*-----
 * Use one of the following Put Record Option (PRO) values when
 * calling TIFPutRecord.
 *-----*/
typedef enum
{
  TIFPRO_STORE_NEW_UNANALYZED_RECORD,
  TIFPRO_ANALYZE_AND_STORE_NEW_RECORD,
  TIFPRO_ANALYZE_AND_UPDATE_RECORD,
  TIFPRO_SPECIAL_HISTORY_FILE_UPDATE,
  TIFPRO_STORE_SPECIAL_ZOMBIE
}
TIF_PUT_RECORD_OPTION;

// Analyze and Store record on disk; set CIG & SKG ties (see tifput.cpp)
// (except UNANALYZED option skips the analysis)
int TIFPutRecord (ILTR_PTRANSL tr, TIF_PUT_RECORD_OPTION nPutOption);

/*-----
 * TIFRetrieveRecord -- read record into a reusable buffer
 *-----*/
int TIFRetrieveRecord
( PSTTIF_TYPE pstTIF,

```

```

        LONG lRecNum,
        ILUT_PBUFFER pRecord );

/*-----
 * Name:      TIFCopyUnmappedFields
 * Purpose:   copy unmapped fields from Original record to Current record.
 *           This is done when sanitizing source records.
 *-----*/
int TIFCopyUnmappedFields(PSTTIF_TYPE pstTIF);

/*-----
 * Name:      TIFCloneRecord
 * Purpose:   make CurrentRecord be an exact copy of OriginalRecord
 *-----*/
int TIFCloneRecord (PSTTIF_TYPE pstTIF);
int TIFCopyRecord (PSTTIF_TYPE pstTIF, ILUT_PBUFFER pFrom, ILUT_PBUFFER pTo);

// get original and current versions of a record
int TIFGetRecord
    ( PSTTIF_TYPE pstTIF,
      LONG lRecNum );          // Record number to retrieve

/*-----
 * Name:      TIFRetrieveFieldByIndex -- get value of numbered field
 *
 * NOTE:   returned LENGTH includes the NULL terminator for non-binary fields.
 *-----*/
int TIFRetrieveFieldByIndex
    ( PSTTIF_TYPE pstTIF,
      INT16 FieldNumber,
      INT32 *plFieldLength,          // OUT: Length of field retrieved
      ILUT_PBUFFER pField,          // Buffer for field retrieved
      TIF_RECORD_VALUE_PTR pRecord ); // record buffer to get field from

// Determine whether field has been altered in the current record
int TIFFFieldChanged
    ( PSTTIF_TYPE pstTIF, IL_PSTR lpszFieldName);

/*-----
 * Name:      TIFGetField -- get value of named field
 *
 * Use nWhich argument to choose ORIGINAL, CURRENT, AUTO, or SOURCE-CACHE
 * value.
 *
 * NOTE:   returned LENGTH includes the NULL terminator for non-binary fields.
 *-----*/
int TIFGetField ( PSTTIF_TYPE pstTIF,
                  IL_PSTR lpszFieldName,
                  int nWhich,
                  LONG *plFieldLength,          // OUT: length of field value
                  ILUT_PBUFFER pField );        // ptr to buffer header

// Get pointer to the value of the view field. NULL ptr if view field is null
int TIFGetViewField (PSTTIF_TYPE pstTIF, IL_PSTR IL_DIST *ppData);

// Compute the number of records that pertain to the current UNLOADING phase
// NOTE: this function does NOT return the count; just computes it and
// puts it in TIF_PertinentRecordCount!!
int TIFComputePertinentRecordCount(PSTTIF_TYPE pstTIF);

// verify that specified record is a valid record that pertains to
// the current UNLOADING PHASE.
int TIFValidateRecord
    ( PSTTIF_TYPE pstTIF, INT32 lRecordNumber );

// position to next record that pertains to the current UNLOADING PHASE.
// (returns TIF_ERR_EOF when we run out of pertinent records)
int TIFPositionToNextRecord (PSTTIF_TYPE pstTIF);

/*-----
 * Name:      TIFOutcome
 * Purpose:   Determine whether current record is to be Replaced, Updated,
 *           Deleted, Added, or Just Left Alone.
 *
 * NOTE:   I would like to switch from past tense "outcomes" to present tense

```

```

*          actions:  Replace, Update, Delete, Add, LeaveAlone...
*-----*/
int TIFGetOutcome (PSTTIF_TYPE pstTIF, INT32 RecordNumber,
                  INT32 IL_DIST *pOutcome);

// Reconcile callback, for keeping track of user choices on what
// to do with current conflicts posted in the Notify dialog. (in tif_ilcr.cpp)

#ifdef __cplusplus
extern "C"    // Make ILCR Callback function (just that 1 function) C-callable
#endif

int IL_DECL TIFReconciliationCallback
( ILTR_PTRANSL  tr,          // tr struct
  IL_PSTR       szLabel,    // Field Name
  IL_PSTR       szData,     // Field Data
  INT32         TargetItem,
  ILT_PTIF IL_DIST * pstTIF ); // Pointer to global info

/*-----
*
* The following is a list of utility functions used for the TIF mechanism.
* These functions can be found in TIFUTIL.C
*-----*/

// Given a Date Time pair in ILIF Date and Time format combine them
// into number of minutes since 1900
INT32 TIFConvertDateTime ( IL_PSTR szDate, IL_PSTR szTime );

// Substitute one substring for another in a string.
int TIFSubstitute
( PSTTIF_TYPE pstTIF,
  ILUT_PBUFFER pMaster,          // ptr to Buffer to alter
  IL_PSTR szReplace,            // Substr to replace
  IL_PSTR szWith );             // Substr to substitute

//----- For Debugging, create formatted dump of
//----- the CURRENTLY OPEN TIF file
int TIFDump (PSTTIF_TYPE pstTIF);

//--- general fieldvalue comparison -- including special case for _repBasic
BOOLEAN TIFFfieldValuesDiffer( PSTTIF_TYPE pstTIF,
                              IL_PSTR pRec1Field,
                              IL_PSTR pRec2Field,
                              INT32 lRec1Field,
                              INT32 lRec2Field,
                              int fieldnum );

//--- general fieldvalue comparison -- NOT including special case for _repBasic
BOOLEAN TIFFfieldValuesDiffer2( ILTR_PTRANSL tr,
                              IL_PSTR pRec1Field,
                              IL_PSTR pRec2Field,
                              INT32 lRec1Field,
                              INT32 lRec2Field,
                              char FieldType,
                              ILTB_ATTRIB FieldAttributes );

/*-----
* Name:      TIFTableCopyAndAdjust
* Purpose:   Copy static table to stack location
*            and adjust it to reflect current Synchronization CR Option
*-----*/
int TIFTableCopyAndAdjust (PSTTIF_TYPE pstTIF);

/*-----
* Name:      TIFTablePickRecordsForSync
* Purpose:   Look up in Table for current cigType and phase to determine
*            what the Original and Current Records are
*-----*/
int TIFTablePickRecordsForSync ( PSTTIF_TYPE pstTIF, INT32 Item,
                                INT16 phase,
                                INT32 *spt,          // IN (3-element array)
                                INT32 *pCurrent,      // OUT
                                INT32 *pOriginal );   // OUT

```

```

/*-----
* Name:      TIFTableGetOutcomeForSync
* Purpose: Look up in Table for current cigType and phase to determine
*           what the Outcome is for current record
*-----*/
int TIFTableGetOutcomeForSync ( PSTTIF_TYPE pstTIF, INT32 Item,
                               INT16 phase,
                               INT32 *pOutcome ); // OUT

int TIFTableGetRawOutcomeForSync ( PSTTIF_TYPE pstTIF,
                                   INT32 Item,
                                   INT16 phase,
                                   INT32 *pOutcome ); // OUT

/*-----
* Name:      TIFBuildSPT
* Purpose: Build itty bitty "cig" and "spt" arrays from CIG members.
*           The resulting arrays are useful for picking out CIG members
*           by ORIGIN.
*-----*/
int TIFBuildSPT ( ILDFX_PHNDL phFile,
                  INT32 Item,
                  INT32 *cig,
                  INT32 *spt );

int TIFRetrieveOrCreatePItem(PSTTIF_TYPE pstTIF, INT32 TargetItem);

//----- Prototype moved from TIFSYNC.CPP to keep CodeWarrior happy.
int TIFDump2 (PSTTIF_TYPE pstTIF);

int dump_and_unmark_cig ( PSTTIF_TYPE pstTIF, INT32 groupNumber,
                          INT32 anchor );

int dump_and_unmark_skg ( PSTTIF_TYPE pstTIF, INT32 groupNumber,
                          INT32 anchor );

/*-----
* Name:      TIFSetRecordNumbers
* Purpose: For a given CIG, figure out which item is "current" and which
*           item is "original".
*
* Called from: TIFGetRecord and TIFSyncGetOutcome
*
* NOTE that Current & Original may be identical!!
*-----*/
int TIFSetRecordNumbers ( PSTTIF_TYPE pstTIF,
                          INT32 Item,
                          INT32 IL_DIST *pCurrent,
                          INT32 IL_DIST *pOriginal );

/*-----
* Name:      TIFSyncFanOutRecurrencePattern
*
* Called by: FindAndUseInstances and iltif.cpp\ILTIFFanItem
*
* NOTE:      caller is always responsible for freeing *ppRepeat if it's set
*
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
*
* WARNING: this function uses TIF_CurrentField and TIF_ExtraField
*-----*/
int TIFSyncFanOutRecurrencePattern ( PSTTIF_TYPE pstTIF,
                                     INT32 ItemToFan,
                                     ILTR_PFANOUT_MAXIMA pMaxima,
                                     TIFSYNC_INSTANCE_TYPE **pInstArray,
                                     ILTR_PREPEAT *ppRepeat,
                                     BOOLEAN bMergeWithOld,
                                     int *pFullCount,
                                     int *pUnexcludedCount );

/*-----
* Name:      TIFSyncDigestFastLoad (in tifsync2.cpp)
*
* Called by: tif.cpp / TIFStartNextPhase at the end of the load-from-source

```



```

    * and load-from-target phases, when doing Fast Sync.
    *-----*/
int TIFSyncDigestFastLoad (ILTR_PTRANSL tr);

/*-----
 * Name:      TIFSyncGetOutcome      (in tifsyntax2.cpp)
 * Called from TIFGetOutcome and ConsiderThisItem
 *-----*/
int TIFSyncGetOutcome ( PSTTIF_TYPE pstTIF,
                        INT32 Item,
                        INT16 phase,
                        INT32 *pOutcome ); // OUT

/*-----
 * Name:      TIFSyncReadHistoryFile  (in tifsyntax2.cpp)
 * Called from TIFStartNextPhase
 *-----*/
int TIFSyncReadHistoryFile(ILTR_PTRANSL tr);

/*-----
 * Name:      TIFSyncCheckOldParameters  (in tifsyntax2.cpp)
 * Called by WrapUpSetUp, which is called by TIFStartNextPhase
 *-----*/
int TIFSyncCheckOldParameters (PSTTIF_TYPE pstTIF, ILDFX_PHNDL phHistory);

//---- "OK To Proceed" function -- in tifoktp.cpp
int TIF_OKToProceed ( ILTR_PTRANSL tr,
                      TIF_OUTCOME_COUNTS *pSrcCounts,
                      TIF_OUTCOME_COUNTS *pTarCounts );

/*-----
 * field value logging functions, in tifutil.cpp
 *-----*/
void TIFLogPutField ( ILTR_PTRANSL tr, IL_PSTR szFldName,
                     UINT8 *pFldData, LONG lFldSize, int rc );

void TIFLogGetField ( ILTR_PTRANSL tr, IL_PSTR szIntro,
                     IL_PSTR szFldName, int nWhich,
                     UINT8 *pFldData, LONG lFldSize, int rc );

void TIFSetHistoryDir ( IL_PSTR szWorkDir,
                       IL_PSTR szILDATA,
                       IL_PSTR szHistoryDir );

/*-----
 * functions shared between iltif.cpp and iltifa.cpp
 *-----*/
int TIFInitStruct (ILTR_PTRANSL tr);

int TIFPutSourceRecord (ILTR_PTRANSL tr);

int TIFGetFieldAttributes
( ILTR_PTRANSL tr,
  IL_PSTR szFieldName,           // IN: Field Name
  INT32 IL_DIST *pMaxLength,     // OUT: Max Field size (or 0 for unlimited)
  char IL_DIST *pFieldType,      // OUT: FldType (ILX_TYPE_XXX)
  ILTB_ATTRIB IL_DIST *pFieldAttributes, // OUT: Field Attribute bits
  IL_PSTR szTypeDesc );         // OUT: copy of szTypeDesc

int TIFGetTargetFieldAttributes
( ILTR_PTRANSL tr,
  IL_PSTR szFieldName,
  INT32 IL_DIST *pMaxLength,     // OUT: Max Field size (or 0 for unlimited)
  char IL_DIST *pFieldType,      // OUT: FldType (ILX_TYPE_XXX)
  ILTB_ATTRIB IL_DIST *pFieldAttributes ); // OUT: Field Attribute bits

int TIFGetInstanceHandle (); // see ILTIFI.CPP

void TIFFreeILTIFBuffers (ILTR_PTRANSL tr);

INT32 TIFKeyDate (PSTTIF_TYPE pstTIF, INT32 Item); //---- in tifxutil.cpp

int TIFMarkAllFigMembers ( ILDFX_PHNDL phFile, INT32 Item,
                           INT32 FlagsToClear, INT32 FlagsToSet,
                           INT32 Flags2ToSet );

```

```
int TIFMergeExclusionLists ( PSTTIF_TYPE pstTIF, INT32 FirstItem,
                           INT32 SItem, INT32 TItem, INT32 PItem );

int TIFCommandeerTargetItem ( PSTTIF_TYPE pstTIF,
                              INT32 PreviousTargetItem,
                              INT32 SourceItem,
                              TIF_RECORD_VALUE_PTR pRec,
                              INT32 *pNewTarget );

int TIFCommandeerSourceItem ( PSTTIF_TYPE pstTIF,
                              INT32 PreviousSourceItem,
                              INT32 TargetItem,
                              TIF_RECORD_VALUE_PTR pRec,
                              INT32 *pNewSource );

IL_PSTR TIFCigName (INT32 cigType);

#endif
```

```
#ifndef TIFSYNC2_H
#define TIFSYNC2_H

/*-----
 * Name:      TIFSYNC2.H
 * Purpose: definitions used by TIF.CPP to access the TIFSYNC2.CPP module
 * Author:   David Boothby, Copyright (c) IntelliLink Corporation, 1995
 *-----*/

extern int ILTIFSyncShouldWeZapIt(ILTR_PTRANSL tr);

extern IL_PSTR ILSYNC_INI_FILE;
extern IL_PSTR ILSYNC_SETTINGS_SECTION;
extern IL_PSTR ILSYNC_HISTORY_NAME;
extern IL_PSTR ILSYNC_SOURCE_PATH;
extern IL_PSTR ILSYNC_TARGET_PATH;
extern IL_PSTR ILSYNC_HISTORY_NAME_HINT;
extern IL_PSTR ILSYNC_EXT_ISH;
extern IL_PSTR ILSYNC_EXT_NEW;
extern IL_PSTR ILSYNC_EXT_OLD;

#endif
```

```

#ifndef _TIFRC_INCLUDED
#define _TIFRC_INCLUDED

/*-----
 * Name:      TIFRC.H
 * Purpose:   TIF Resources
 *
 * This file should NOT be included by any modules
 * outside of the ILTIF subsystem.
 *
 * Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
 *-----*/

// Variables needed for the string table
#define TIF_STR_TEXT 301
#define TIF_STR_UNSPECIFIED 302
#define TIF_STR_RECONCILING 303
#define TIF_STR_REPSTART_PSEUDO_FLDNAME 304
#define TIF_STR_REPSTOP_PSEUDO_FLDNAME 305
#define TIF_STR_REPEX_PSEUDO_FLDNAME 306
#define TIF_STR_EXCLUSIONS 307
#define TIF_STR_RECURRING 308
#define TIF_STR_MULTIDAY 309
#define TIF_STR_TRUE 310
#define TIF_STR_FALSE 311

//----- string IDs 310-349 reserved for TIFOKTP (see tifoktp.h)

#define TIF_STR_CIG001 350
#define TIF_STR_CIG010 351
#define TIF_STR_CIG011 352
#define TIF_STR_CIG012 353
#define TIF_STR_CIG100 354

//----- the next ten #defines have cousins defined below (in lockstep)
#define TIF_STR_CIG101 355
#define TIF_STR_CIG102 356
#define TIF_STR_CIG110 357
#define TIF_STR_CIG210 358
#define TIF_STR_CIG111 359
#define TIF_STR_CIG112 360
#define TIF_STR_CIG211 361
#define TIF_STR_CIG212 362
#define TIF_STR_CIG213 363
#define TIF_STR_CIG132 364

#define TIF_STR_EOL 369
#define TIF_STR_DIVIDER 370
#define TIF_STR_SUMMARY 371
#define TIF_STR_CLASH_COUNT 372
#define TIF_STR_ADD_COUNT 373
#define TIF_STR_CHG_COUNT 374
#define TIF_STR_DEL_COUNT 375
#define TIF_STR_NOCHG_COUNT 376

#define TIF_STR_NON_UNIQUE_ID 380

//----- degrees of separation between cousins
#define TIF_MEX_COUSIN_ADDEND 30

//----- the next ten #defines must all be exactly +30 each compared to
//----- the ten cousins above (355-364) -- see TIFDUMP.CPP
#define TIF_STR_CIG101MEX 385
#define TIF_STR_CIG102MEX 386
#define TIF_STR_CIG110MEX 387
#define TIF_STR_CIG210MEX 388
#define TIF_STR_CIG111MEX 389
#define TIF_STR_CIG112MEX 390
#define TIF_STR_CIG211MEX 391
#define TIF_STR_CIG212MEX 392
#define TIF_STR_CIG213MEX 393
#define TIF_STR_CIG132MEX 394

#endif

```

```

/*-----
* Name:      ILTIF.RC
* Purpose:   RC file for the TIF mechanism.
* Sections:
*           String Table
*
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/

#ifdef USE_NEW_ILCR

    #include "ilcr2.rc"

#endif

#include "tifrc.h"

#ifndef USE_NEW_ILCR

    #include "ilcr.rc"

#endif

#include "iltrerr.h"
#include "iltrerr.rc"
#include "iltrmsg.rc"
#include "ilver.rc"

STRINGTABLE
BEGIN
    TIF_STR_TEXT,          "<NO VALUE>"
    TIF_STR_UNSPECIFIED,   "<Unspecified>"
    TIF_STR_RECONCILING,   "Reconciling Data Items" // progress bar caption
    TIF_STR_REPSTART_PSEUDO_FLDNAME, "First Date" // pseudo-fieldname in ILCR
    TIF_STR_REPSTOP_PSEUDO_FLDNAME, "Last Date" // pseudo-fieldname in ILCR
    TIF_STR_REPEX_PSEUDO_FLDNAME, "Excluded Dates" // pseudo-fieldname in ILCR
    TIF_STR_EXCLUSIONS,   "\001 and \001 others" // pseudo-fieldvalue in ILCR
    TIF_STR_RECURRING,      "repeating"
    TIF_STR_MULTIDAY,       "multi-day"
    TIF_STR_TRUE,           "Yes" // "Yes" preferred to "True"
    TIF_STR_FALSE,          "No" // "No" preferred to "False"
    TIF_STR_NON_UNIQUE_ID,  "\r\n      => ID \042%.200s\042 is not unique."
END

//STRINGTABLE
//BEGIN
//    TIF_STR_SYNC_SUMMARY,  "\015\012\015\012Summary of Data Analysis:\015\012\015\012"
//    TIF_STR_SYNC_CONFLICT_COUNT, "Conflicting Changes: \001\015\012"
//    TIF_STR_SYNC_CHG_COUNT,      "Non-conflicting Changes: \001\015\012"
//    TIF_STR_SYNC_ADD_COUNT,       "Additions: \001\015\012"
//    TIF_STR_SYNC_DEL_COUNT,       "Deletions: \001\015\012"
//    TIF_STR_SYNC_SAME_COUNT,      "Records unchanged: \001\015\012"
//END

//----- Note position reversal of First/Second Systems, between the
//----- names and values of the following strings:
STRINGTABLE
BEGIN
    TIF_STR_CIG001, "Added on First System\015\012"
    TIF_STR_CIG010, "Deleted from Both Systems\015\012"
    TIF_STR_CIG011, "Deleted from Second System\015\012"
    TIF_STR_CIG012, "Deleted from Second System, Changed on First\015\012"
    TIF_STR_CIG100, "Added on Second System\015\012"
    TIF_STR_CIG101, "Same Record Exists on Both Systems\015\012"
    TIF_STR_CIG102, "Added differently on Each System\015\012"
    TIF_STR_CIG110, "Deleted from First System\015\012"
    TIF_STR_CIG210, "Deleted from First System, Changed on Second\015\012"
    TIF_STR_CIG111, "Unchanged on Both Systems\015\012"
    TIF_STR_CIG112, "Changed on First System\015\012"
    TIF_STR_CIG211, "Changed on Second System\015\012"
    TIF_STR_CIG212, "Changed identically on Both Systems\015\012"
    TIF_STR_CIG213, "Changed differently on Each System\015\012"
    TIF_STR_CIG132, "Conflict Resolved by Compromise\015\012"
END

```

```

STRINGTABLE
BEGIN
    TIF_STR_CIG101MEX, "Exclusion Lists Differ\015\012"
    TIF_STR_CIG102MEX, "TIF_STR_CIG102MEX\015\012"
    TIF_STR_CIG110MEX, "TIF_STR_CIG110MEX\015\012"
    TIF_STR_CIG210MEX, "TIF_STR_CIG210MEX\015\012"
    TIF_STR_CIG111MEX, "Exclusion List(s) Changed\015\012"
    TIF_STR_CIG112MEX, "Changed on First System and Exclusion Lists Differ\015\012"
    TIF_STR_CIG211MEX, "Changed on Second System and Exclusion Lists Differ\015\012"
    TIF_STR_CIG212MEX, "Changed on Both Systems and Exclusion Lists Differ\015\012"
    TIF_STR_CIG213MEX, "Changed on Each System and Exclusion Lists Differ\015\012"
    TIF_STR_CIG132MEX, "Conflict Resolved, Exclusion Lists Differ\015\012"
END

```

```

STRINGTABLE
BEGIN
    TIF_STR_EOL, "\015\012"
    TIF_STR_DIVIDER, "- - - - -"
    TIF_STR_SUMMARY, "\015\012Summary of Data Analysis:\015\012\015\012"
    TIF_STR_CLASH_COUNT, "Records Added or Changed - Conflicting: \001\015\012"
    TIF_STR_ADD_COUNT, "Records Added - Not Conflicting: \001\015\012"
    TIF_STR_CHG_COUNT, "Records Changed - Not Conflicting: \001\015\012"
    TIF_STR_DEL_COUNT, "Records Deleted: \001\015\012"
    TIF_STR_NOCHG_COUNT, "Records Unchanged: \001\015\012"
END

```

```

#ifdef USE_NEW_ILCR
    #include "tifoktp.rc"
#endif

```

```

/*-----
* Name:      ILTIF.CPP (and ILTIFA.CPP and ILTIFI.CPP)
* Purpose:   These 3 modules contain almost all of the public entrypoints
*           into the ILTIF mechanism.
*
* Functions (local functions are indented):
*
*           ILTIFCreate -- same as ILTIFInit
*           ILTIFInit  -- "call me first" -- top-level initializer
*           TIFInitStruct -- called by ILTIFInit
*
*           ILTIFHowManyField -- please don't call this function!!
*           ILTIFGetFieldName -- please don't call this function!!
*           ILTIFDefField  -- used to tell TIF about a field
*           ILTIFDefFieldEx -- extended version of ILTIFDefField
*           DefineField    -- internal DefField function
*           ILTIFDefFieldN -- tell TIF about a field by ILTR field#
*           TIFGetFieldAttributes -- guts of ILTIFGetFieldAttributes
*           ILTIFGetFieldAttributes -- analogue to ILTR\ILFldTypeEx
*           TIFGetTargetFieldAttributes -- restricted variant of previous fn
*           ILTIFFillFields -- loop thru field list calling ILTIFDefField
*
*           ILTIFInitRecord -- called from iltr\export.c for ILX_V4 mode
*           ILTIFPutField  -- put field value into current record in TIF
*           ILTIFPutFieldEx -- special version of ILTIFPutField
*           ILTIFPutFieldEx2 -- OBSOLETE: another version of ILTIFPutField
*           ILTIFPutRecord
*           ILTIFWriteRecord -- same as ILTIFPutRecord
*           ILTIFCloneRecord
*
* Author:   Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*           David Boothby, Copyright (c) IntelliLink Corp., 1995
*-----
*
* Games that we play, to integrate Field Mapping into TIF, and to accommodate
* unmapped and "extraneous" fields (i.e. fields that ILTR knows nothing about).
*
* Relevant data structures are:
*
* 1, The TIF Field List, which lists all the Field Names. This is
* based largely on the TARGET APP's fieldlist. It is up to the
* Target App Translator to tell TIF about its fields, including all
* MAPPED fields, and optionally UNMAPPED fields as well -- at said
* translator's discretion.
*
* When doing "SmartMerge", that's all there is to say -- just Target
* App Fields. But when doing Synchronization, the TIF Fieldlist may
* contain unmapped Source App fields in addition to Target App fields.
*
* Unmapped Source App Fields are distinguished from other fields by
* having their names prefixed by the character '\01' (CTRL-A).
*
* 2. The TIF Cache, which is only used when data being loaded into TIF needs
* to be subjected to Field Mapping.
*
* Functions which get involved in Field Mapping Games are:
*
* a. ILTIFDefField
* b. ILTIFPutField
* c. ILTIFPutRecord
* f. ILTIFGetField
*
* and the ILTR functions ILFldPut and ILFldGet.
*
* Details for each of these functions are as follows:
*
* a. ILTIFDefField:
*
* Normally this just results in adding another Field Desc entry to the
* TIF Field list. But when we're getting Field definitions from
* the Source Translator (which is only done for Synchronization, and
* happens when ILTR_phase == ILTR_PHASE05) we check, on each ILTIFDefField
* call, to see if the field is mapped. If it is, we IGNORE it!! If it
* isn't mapped, we add it to the Field List with a '\01' prefix.
*

```

* b. ILTIFPutField:

*
 * Normally this just results in putting field data into the current
 * Record Value structure. But when we are loading Source App data, which
 * is done while ILTR_phase == ILTR_PHASE20, we check, on each ILTIFPutField
 * call, to see if the field is mapped. If it is, we put the field name
 * and value into the ILTIF Cache (from which ILTIFPutRecord may eventually
 * fetch it). If the field is NOT mapped, we lookup the '\01'-prefixed
 * field name in the TIF Field List, and if found, we put the field data into
 * the current Record Value structure.

* c. ILTIFPutRecord:

*
 * Normally this just results in writing a filled-in Record Value structure
 * into the TIF file. But when we are loading Source App data, (which is
 * done while ILTR_phase == ILTR_PHASE20) we have to do Field Mapping first.
 * Source App data that has been supplied for UNMAPPED Source App fields is
 * already in the Record Value structure. But Source App data that has been
 * supplied for MAPPED fields is in the ILTIF Cache at this point. So the
 * ILTIFPutRecord function loops through the entire TIF Field List, calling
 * the ILTR function ILFldGet for each MAPPED field that it knows about.

* d. ILTR ILFldGet:

*
 * This is the "Shining Path" to IntelliLink Field Mapping.
 * This function has a traditional use, for ILIF-based translation, where
 * the TARGET Translator calls ILFldGet to get MAPPED field values from
 * ILIF. In the "ILX_V4" world, that traditional use for ILFldGet is no
 * longer exercised. For ILX_V4, ILFldGet is called two ways:

- * i. When ILTIFPutRecord needs to do Field Mapping, it calls ILFldGet.
 * (This is when we're loading Source App data into TIF - PHASE20.)
- * ii. When ILTIFGetField needs to do Field Mapping, it calls ILFldGet.
 * (This is when we're unloading from TIF to SOURCE APP - PHASE40.)

* Note that for both of these cases where ILTIF calls ILFldGet, ILFldGet
 * turns right around and calls ILTIFGetField. This may seem absurd,
 * especially when ILTIFGetField has just called ILFldGet, but it actually
 * works out quite nicely.

* Example: xlator calls ILTIFGetField, seeking value for
 * fieldname="SourceAppZipcode". ILTIFGetField calls
 * ILFldGet, which sees that TargetAppZipcode is mapped to
 * SourceAppZipcode. So it then calls ILTIFGetField to get
 * the value for fieldname="TargetAppZipcode", and eventually
 * passes the value back...

* Of course ILTIFGetField allows only one level of recursion.

* e. ILTR ILFldPut:

*
 * Nothing interesting here. For ILX_V4 calls to ILFldPut simply turn
 * into calls to ILTIFPutField.

* f. ILTIFGetField:

*
 * There are 4 distinct circumstances for calls to ILTIFGetField:

- * i. When we are Unloading TIF to the Target App, either in ILX_V3
 * mode or in ILX_V4 mode with ILTR_phase==ILTR_PHASE30, the
 * Target Translator calls ILTIFGetField for each Target Field,
 * and we simply get the requested field from the TIF database
 * and pass it back. No games to play. No field mapping.
- * ii. When we are LOADING Source App Data directly into TIF, and
 * doing Field Mapping ensue. This never happens in ILX_V3
 * mode, but in ILX_V4 mode it happens during ILTR_PHASE20.
 * In this circumstance, the call stack looks like this:

*
 * xlator calls ILTIFPutRecord...
 * which in turn calls ILFldGet (N times, in a loop),
 * seeking values for Target App Fields...
 * which decipheres the Field Map, and calls ILTIFGetField
 * to get values from the corresponding Source App Field(s)...


```

*           which gets those values from the ILTIF Cache!!!
*
*   iii. When we are UNLOADING TIF data to the Source App (which only
*         happens when doing Synchronization -- in ILTR_PHASE40).
*         The SOURCE xlator calls ILTIFGetField for each Source App
*         field that it cares about. ILTIFGetField tries to satisfy
*         each request as follows:
*
*         * First it looks to see if there is an UNMAPPED SOURCE APP
*           FIELD (with '\01'-prefixed fieldname). If so, it returns
*           the value of said field.
*
*         * Failing that, it then calls ILFldGet to have Field Mapping
*           done....
*
*   iv. Continuing scenario (iii), the ILFldGet call resolves to one
*        or more ILTIFGetField calls, which are now looking for
*        TARGET APP Fields. No problem...this is like a re-run of
*        scenario (i). We grab the requested value from TIF and
*        pass it back.
*
*-----*/
#include "iltr.h"

#define EXIT_WITH_ERROR(e) {rc=e; goto Exit;}
#define LOG_ERR_AND_EXIT(e1, e2) EXIT_WITH_ERROR(ILERROR(e1, e2))

static int DefineField
( ILTR_PTRANS� tr,
  IL_PSTR szFldName,
  LONG lFldSize,
  char IL_DIST *pFldType, // ptr to 1 char
  IL_PSTR szFormat,
  ILTB_ATTRIB FieldAttributes,
  INT32 *pExtraAttributes, // IN: array of 1 or more...
  IL_PSTR szDefault,
  ILTR_NDX iltrFieldnum );

static int GetFieldGuts ( ILTR_PTRANS� tr,
                          IL_PSTR szFieldName,
                          int nWhich,
                          BOOLEAN bMapCharsIfNonBinary,
                          LONG *pFieldLength,
                          IL_PANY *pFieldValue );

static int MapCharsFromIL_IfNonBinary ( ILTR_PTRANS� tr,
                                         IL_PSTR FieldName,
                                         ILUT_PBUFFER pField,
                                         INT32 *pFieldLength );

static int MarkFlag2ForAllCIGMembers ( ILDFX_PHNDL phFile, INT32 Item,
                                       INT32 FlagsToSet );

/*-----
* The following entrypoint is obsolete, but will be retained for awhile to
* avoid nuisance revlocks between TIF DLL and translators build with ILTR
* libraries including FLDPUT.OBJ from 1/3/96 through 1/19/96.
*-----*/
extern "C" TIF_DLL_ENTRYPOINT ILTIFPutFieldEx2
( ILTR_PTRANS� tr,
  IL_PSTR szFldName,
  IL_PANY pFldData,
  LONG lFldSize,
  BOOLEAN bLengthGuaranteed, // see NOTE above
  BOOLEAN bSpecialFanningAdjustment,
  BOOLEAN bMapCharsIfNonBinary );

/*-----
* Name:      ILTIFCreate
* Context:   When using ILTIF for SmartMerge
* Purpose:   Initialize TIF, create TIF file, etc.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFCreate ( ILTR_PTRANS� tr,
                                IL_HINST hObsoleteUnusedArgument,

```

```

                                LONG lFields )
{
    int rc = ILTIFInit(tr, NULL, lFields);
    return rc;
} //---- ILTIFCreate

/*-----
* Name:      ILTIFInit
* Purpose: Initialize TIF, open or create TIF file, etc...
*
* Called by: ILTIFCreate and possibly by users of ILTIF
*
* NOTE:      SUCCESS means we've allocated TIF structures and opened files.
*            anything else means NOTHING is allocated and NO FILE are OPEN.
*            So if ILTIFInit fails, no cleanup is required.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFInit ( ILTR_PTRANSL tr,
                                IL_PSTR lpszTIFFilename, // unused!!
                                LONG lFields )
{
    int rc = TIFInitStruct(tr);
    if (rc != SUCCESS)
        return rc;

    /*-----
    * make sure we start out with a legal field count. The exact number
    * is unimportant; we just allocate space for N fields, and we can
    * grow the space if we need room for more.
    *-----*/
    if (lFields == 0)
        lFields = TIF_FIELD_COUNT_INCREMENT;

    // Initialize the mechanism
    rc = TIFInit (&ILTIF_pstTIF, tr, (int) lFields);
    if (rc != SUCCESS)
    {
        IL_FREE (ILTIF_hstILTIF, ILTR_pILTIF);
        ILTR_pILTIF = NULL;
    }

    return rc;
} //---- ILTIFInit

/*-----
* Name:      TIFInitStruct
* Purpose: Initialize ILTIF structure
*
* Called by: ILTIFInit and ILTIFSyncInit
*-----*/
int TIFInitStruct (ILTR_PTRANSL tr)
{
    int rc;

    rc = TIFGetInstanceHandle (); // see ILTIFI.CPP
    if (rc != SUCCESS)
        return rc;

    //---- set signature for Heap Logging (if enabled)
    ILUT_MemSig ("ILTIF");

    // Allocate an initial global info record
    IL_HANDLE hstILTIF; // Handle to global data
    ILTR_pILTIF = (ILT_PILTIF) IL_ALLOC (sizeof(ILT_ILTIF), hstILTIF);
    if (ILTR_pILTIF == NULL)
        return ILERROR_L ((INT32) sizeof(ILT_ILTIF), TIF_ERR_MEM);

    IL_MEMSET (ILTR_pILTIF, 0, sizeof(ILT_ILTIF));
    ILTIF_hstILTIF = hstILTIF;

    /*-----
    * When running under older apps the ILDATA directory is hardcoded,

```

```

    * but newer apps put the name in ILTR_szUserDir.
    *-----*/
IL_PSTR lpszILDATA;
if (ILTR_VERSION_IS_AT_LEAST(29))
    lpszILDATA = ILTR_szUserDir;
else
    lpszILDATA = "";

TIFSetHistoryDir (ILTR_szCurWD, lpszILDATA, ILTIF_szHistoryDir);

/*-----
 * Initialize reusable field buffers.
 *-----*/
rc = ILUT_InitBuffer ( &ILTIF_Field, TIF_INITIAL_FIELD_BUF_SIZE,
                      TIF_BUF_INC, ILTR_MAX_FIELDLNGTH+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &ILTIF_Field2, TIF_INITIAL_FIELD_BUF_SIZE,
                          TIF_BUF_INC, ILTR_MAX_FIELDLNGTH+1 );

if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &ILTIF_Field3, TIF_INITIAL_FIELD_BUF_SIZE,
                          TIF_BUF_INC, ILTR_MAX_FIELDLNGTH+1 );

if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &ILTIF_TmpBuf, TIF_INITIAL_FIELD_BUF_SIZE,
                          TIF_BUF_INC, ILTR_MAX_FIELDLNGTH+1 );

if ((rc == SUCCESS) && (ILTR_pTmpBuf == NULL))
{
    //---- initialize reusable buffer ILTR_pTmpBuf
    ILTR_pTmpBuf = (ILUT_PBUFFER)
        IL_ALLOC (sizeof(ILUT_BUFFER), ILTR_hTmpBuf);
    if (ILTR_pTmpBuf == NULL)
        rc = ILTR_ERR_NOMEM;
    else
        rc = ILUT_InitBuffer (ILTR_pTmpBuf, 0, 512, ILTR_MAX_FIELDLNGTH+1);
}

if (rc != SUCCESS)
{
    TIFFreeILTIFBuffers (tr);
    return ILERROR(rc, TIF_ERR_MEM);
}

//---- turn off test mode to use TIF in earnest...
ILTR_pILTIF->bTestMode = FALSE;

return SUCCESS;
} //---- TIFInitStruct

/*-----
 * Name:      ILTIFHowManyField
 * Purpose: Returns how many fields are currently defined in TIF
 * Input:     tr
 *            tr struct
 *            lNumOfFlds
 *            Pointer to number of fields defined
 * Return:    SUCCESS      - If all went well.
 *            TIF_ERR_MEM  - If there is no global structure
 *            lNumFields filled in
 * Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
 * Notes:
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFHowManyField ( ILTR_PTRANSL tr,
                                       LONG *pNumOfFlds )
{
    if (ILTR_phase == ILTR_PHASE40)
    {
        //----- for Phase 40 field mapping applies; don't really
        //----- want the raw TIF field count...
        UINT uFieldCount = ILFldCount(tr);
        *pNumOfFlds = (LONG) uFieldCount;
        return SUCCESS;
    }
}

```

```

    }
    else
        return TIFHowManyField(&ILTIF_pstTIF, pNumOfFlds);
} //---- ILTIFHowManyField

/*-----
* Name:      ILTIFGetFieldName
* Purpose: Returns field name given an index
* Input:   tr
*          tr struct
*          lFldNdx
*          Field index
*          szFldName
*          Buffer for name of field
* Return:  SUCCESS      - If all went well.
*          TIF_ERR_MEM   - If there is no global structure
*          szFldName filled in
* Author:   Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:
*-----*/
TIF_DLL_ENTRYPOINT ILTIFGetFieldName ( ILTR_PTRANSI tr,
                                       LONG lFldNdx,
                                       IL_PSTR szFldName )
{
    int i = (int) lFldNdx;

    //-----this Phase40 stuff is probably bogus...
    //if (ILTR_phase == ILTR_PHASE40)
    //{
    //    //----- Get field name and check if mapped
    //    int bMapped = ILFldNameByIndex (tr, i, szFldName, ILTR_MAX_FLDNAME);
    //    if (bMapped == -1)
    //    {
    //        IL_MAKE_STRING_NULL(szFldName);
    //        return TIF_ERR_BAD_FLDNAME;
    //    }
    //    else
    //        return SUCCESS;
    //}
    //else
    {
        int rc = TIFGetFieldName (&ILTIF_pstTIF, i, szFldName);
        return rc;
    }
} //---- ILTIFGetFieldName

/*-----
* Name:      ILTIFDefField
* Purpose: Define a field in the TIF database.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFDefField ( ILTR_PTRANSI tr,
                                   IL_PSTR szFldName,
                                   LONG lFldSize,
                                   char IL_DIST *pFldType, // ptr to 1 char
                                   IL_PSTR szFormat,
                                   ILTB_ATTRIB FieldAttributes,
                                   IL_PSTR szDefault )
{
    INT32 extra=0;
    int rc = DefineField ( tr, szFldName, lFldSize, pFldType, szFormat,
                          FieldAttributes, &extra, szDefault, TIF_NOTSET );
    return rc;
} //---- ILTIFDefField

/*-----
* Name:      ILTIFDefFieldEx
* Purpose: Define a field in the TIF database -- extended API
*-----*/
TIF_DLL_ENTRYPOINT ILTIFDefFieldEx

```

```

        ( ILTR_PTRANSL tr,
          IL_PSTR szFldName,
          LONG lFldSize,
          char IL_DIST *pFldType, // ptr to 1 char
          IL_PSTR szFormat,
          ILTB_ATTRIB FieldAttributes,
          INT32 *pExtraAttributes, // IN: array of 1 or more...
          IL_PSTR szDefault )
    {
        int rc = DefineField ( tr, szFldName, lFldSize, pFldType, szFormat,
                              FieldAttributes, pExtraAttributes, szDefault,
                              TIF_NOTSET );

        return rc;
    } //---- ILTIFDefFieldEx

/*-----
* Name:      DefineField
* Purpose:   Define a field in the TIF database -- internal function
*-----*/
static int DefineField
( ILTR_PTRANSL tr,
  IL_PSTR szFldName,
  LONG lFldSize,
  char IL_DIST *pFldType, // ptr to 1 char
  IL_PSTR szFormat,
  ILTB_ATTRIB FieldAttributes,
  INT32 *pExtraAttributes, // IN: array of 1 or more...
  IL_PSTR szDefault,
  ILTR_NDX iltrFieldnum )
{
    int rc;
    char szRelFldName[ILTR_MAX_FLDNAME]; // Name of the relative field
    char szUSFName[ILTR_MAX_FLDNAME+1]; // Decorated UNMAPPED Source Field Name
    IL_PSTR lpszFldName;
    BOOLEAN bPositive;
    PSTTIF_TYPE pstTIF;
    TIF_FIELDLIST_PTR pFieldList;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    pstTIF = &ILTIF_pstTIF;

    if (ILTR_phase != TIF_PreviousILTRPhase)
    {
        ILTIFlogszul("\r\n---- Starting ILTR phase %ld", (UINT32) ILTR_phase);
        TIF_PreviousILTRPhase = ILTR_phase;
    }

    //--- treat NULL the same as zero-length string. Don't GPF.
    if (pFldType == NULL) pFldType = "";
    if (szFldName == NULL) szFldName = "";
    if (szDefault == NULL) szDefault = "";

    if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 40))
    {
        char szBuf[200];
        IL_SPRINTF ( szBuf, "ILTIFDefFieldEx.%c.%lx.%08lx %s (default=%s)",
                    pFldType[0], pExtraAttributes[0], FieldAttributes,
                    szFldName, szDefault );
        ILTIFlogsz(szBuf);
    }

    int nFieldType; nFieldType = pFldType[0];
    switch (nFieldType)
    {
        case ILX_TYPE_TEXT:
        case ILX_TYPE_BOOL:
        case ILX_TYPE_DATE:
        case ILX_TYPE_NUMBER:
        case ILX_TYPE_PHONE:
        case ILX_TYPE_TIME:
        case ILX_TYPE_BINARY:

```

```

        break;

    default:
        return TIF_ERR_BAD_FIELD_TYPE;
}

//---- make default assumptions about Related Fields...
IL_MAKE_STRING_NULL(szRelFldName);
bPositive = TRUE;

if (ILTR_phase == ILTR_PHASE05)
{
    /*-----
    * We are being told about SOURCE APP FIELDS. We handle MAPPED
    * and UNMAPPED source app fields very differently. The MAPPED
    * ones go into a separate Field List. The UNMAPPED ones go
    * into the main field list, prefixed with '\01'.
    *-----*/
    if (pExtraAttributes[0] & TIFEA_ISNT_MAPPED)
    {
        //----- UNMAPPED SOURCE FIELDS are decorated & put in main list
        IL_SPRINTF(szUSFName, TIF_USFN_FORMAT, szFldName);
        lpszFldName = szUSFName; // use Decorated Field Name
        pFieldList = TIF_pFieldList;
    }
    else
    {
        //----- MAPPED SOURCE FIELDS go into special list
        lpszFldName = szFldName; // use un-decorated Field Name
        pFieldList = TIF_pSourceFieldList;
    }
}
else
{
    //---- ALL Target Fields go into the main list
    pFieldList = TIF_pFieldList;
    lpszFldName = szFldName; // use un-decorated Field Name

    if (!ILTR_pILTIF->bTestMode)
    {
        // Get the relative field (Test Mode can't handle this!)
        bPositive = FALSE;
        rc = ILFldAssoc (tr, szFldName, szRelFldName, &bPositive);
        // ignore error here ... it's perfectly normal for this call to fail!!
        // if (nTIFReturn != SUCCESS)
        // return nTIFReturn;
    }
}

if (pExtraAttributes[0] & TIFEA_ISNT_MAPPED)
{
    /*-----
    * Unmapped fields must not be used for reconciliation!!
    * Turn off Start&End Date/Time attribs because unmapped fields
    * cannot be used for appointment overlap checking.
    *-----*/
    FieldAttributes |= ILTB_ATT_NO_RECONCILE;
    FieldAttributes &= ~( ILTB_ATT_KEY_FIELD
        | ILTB_ATT_STARTDATETIME
        | ILTB_ATT_ENDDATETIME );
}

rc = TIFDefineOneField ( pstTIF, pFieldList,
    lpszFldName,
    lFldSize,
    nFieldType,
    szFormat,
    szRelFldName,
    FieldAttributes,
    pExtraAttributes,
    bPositive,
    szDefault,
    iltrFieldnum );

return rc;

```

```

} //---- DefineField

/*-----
 * ILTIFDefFieldN
 * Purpose: Define a field in the TIF database by specifying ILTR fieldnum.
 *
 * This function knows all the quirky rules about mapped/unmapped status of
 * various SPECIAL fields, and it knows the difference between
 * ILTR_UNMAPPED_BUT_TAGGED and ILTR_UNMAPPED.
 *
 * NOTE:  this function replaces iltr\ilrtif.c\ILTRTIFDefFieldN.
 *        this function is more powerful than ILTRTIFDefFieldN.
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFDefFieldN // Tell TIF about a field
( ILTR_PTRANS tr,
  ILTR_NDX fldnum,          // field# in full field list
  INT32 ExtraAttributes )
{
    int rc;

    ILTR_FLDPTR fldlst;
    ILTR_NDX fldmax;

    //----- Complain if running under a stone age app or engine
    if (ILTR_VERSION_IS_PRIOR_TO (14))
        return ILERROR(ILTR_version, ILTR_ERR_STONE_AGE_APP);

    if (ILTR_direction == ILTR_EXPORT)
    {
        fldlst = ILTR_map.pSource;
        fldmax = ILTR_map.nSource + ILTR_pTableInfo->nExtraFields;
    }
    else
    {
        fldlst = ILTR_map.pTarget;
        fldmax = ILTR_map.nTarget + ILTR_pTableInfo->nExtraFields;
    }

    //---- squawk if caller supplies absurd field number
    if (fldnum < 0 || fldnum >= fldmax)
        return ILERROR ((int) fldnum, TIF_ERR_BAD_FIELD_NUM);

    //---- don't allow user to tell us whether or not field is mapped
    ExtraAttributes &= ~(TIFEA_IS_MAPPED | TIFEA_ISNT_MAPPED);

    /*-----
     * For hidden fields we apply special rules.  "Vanilla" hidden
     * fields are UNMAPPED, but certain "special" hidden fields are
     * implicitly mapped, even though they are marked UNMAPPED in
     * the ILTR field list.
     *-----*/
    if (fldlst[fldnum].Attribs & ILTB_ATT_HIDDEN_FIELD)
    {
        if ( (IL_STRINGS_EQUAL(fldlst[fldnum].IntName, ILTR_REP_BASIC))
            || (IL_STRINGS_EQUAL(fldlst[fldnum].IntName, ILTR_REP_XDATE))
            || (IL_STRINGS_EQUAL(fldlst[fldnum].IntName, ILTR_SUB_TYPE))
            || (IL_STRINGS_EQUAL(fldlst[fldnum].IntName, ILTR_APP_DATA)) )
            ExtraAttributes |= TIFEA_IS_MAPPED;
        else
            ExtraAttributes |= TIFEA_ISNT_MAPPED;
    }

    else if (fldlst[fldnum].MapField == ILTR_UNMAPPED_BUT_TAGGED)
    {
        /*-----
         * Almost everyone in the world is duped into thinking that
         * UNMAPPED_BUT_TAGGED fields are actually MAPPED.  But TIF
         * needs to know the truth that such fields are NOT mapped.
         *-----*/
        ExtraAttributes |= TIFEA_ISNT_MAPPED;
    }

    else if (fldlst[fldnum].MapField != ILTR_UNMAPPED)
        //---- This field is Mapped, pure & simple
        ExtraAttributes |= TIFEA_IS_MAPPED;
}

```

```

else
{
    /*-----
    * This field isn't mapped. But maybe it's a combined field
    * and one of its sub-items is mapped...
    *-----*/
    BOOLEAN bMapped = FALSE;
    int j;

    for (j = fldnum+1; j < fldmax; j++)
    {
        /*----- If no mapped sub-items found, the field remains unmapped
        if (fldlst[j].ItemNo == 1)
            break;

        /*----- If we find a mapped sub-item, mark the field as mapped
        else if ( (fldlst[j].MapField != ILTR_UNMAPPED)
            && (fldlst[j].MapField != ILTR_UNMAPPED_BUT_TAGGED) )
        {
            bMapped = TRUE;
            break;
        }
    }

    ExtraAttributes |= (bMapped ? TIFEA_IS_MAPPED : TIFEA_ISNT_MAPPED);
}

rc = DefineField ( tr,
    fldlst[fldnum].IntName,
    fldlst[fldnum].Width,
    &fldlst[fldnum].Type,
    NULL,
    fldlst[fldnum].Attribs,
    &ExtraAttributes,
    fldlst[fldnum].TypeDesc,
    fldnum );

return rc;
} //---- ILTIFDefFieldN

/*-----
* Name:      TIFGetFieldAttributes
* Purpose:   Get parts of a field definition that is stored in the TIF database
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*
* NOTE:     this function tries to be very clever; be sure it isn't too clever
*           for your needs. It tries to get you attribute info about one of
*           "YOUR" fields, no matter who YOU are. Depending on what ILTR PHASE
*           we're in, and whether Field Mapping is a factor, we may get the
*           info from TIF itself, or from ILTR.
*
* We always ask TIF first, because it's always possible for translators to
* define fields in TIF that don't exist in the ILTR field list.
*-----*/
int TIFGetFieldAttributes
( ILTR_PTRANSL tr,
  IL_PSTR szFieldName,          // IN: Field Name
  INT32 IL_DIST *pMaxLength,    // OUT: Max Field size (or 0 for unlimited)
  char IL_DIST *pFieldType,     // OUT: FldType (ILX_TYPE_XXX)
  ILTB_ATTRIB IL_DIST *pFieldAttributes, // OUT: Field Attribute bits
  IL_PSTR szTypeDesc )         // OUT: copy of szTypeDesc
{
    int rc;
    char szUSFName[ILTR_MAX_FLDNAME+1];
    IL_PSTR name;
    BOOLEAN bSourceFieldWanted = ( (ILTR_phase == ILTR_PHASE20)
                                   || (ILTR_phase == ILTR_PHASE40));
    if (bSourceFieldWanted)
    {
        /*-----
        * Given that TIF's field info is organized according to the TARGET
        * app's field list, looking for attributes of a SOURCE field is a
        * bit tricky. We first look for the decorated name of an UNMAPPED
        * source field.
        */
    }

```



```

    /*-----*/
    IL_SPRINTF(szUSFName, TIF_USFN_FORMAT, szFieldName);
    name = szUSFName;    // decorated name for unmapped source field
}
else
    /*--- no field mapping issue; just look up undecorated name in TIF
    name = szFieldName;

TIF_FIELD_DESC_PTR pDefinition;
rc = TIFLookUpFieldDefinition (&ILTIF_pstTIF, name, &pDefinition);
if (rc == SUCCESS)
{
    *pMaxLength          = pDefinition->MaxLength;
    *pFieldType          = pDefinition->FieldType;
    *pFieldAttributes    = pDefinition->FieldAttributes;
    if (szTypeDesc != NULL)
        IL_SAFE_STRINGCOPYN ( szTypeDesc, pDefinition->szDefault,
                               ILTR_MAX_TYPEDESC );
}
else if (rc == TIF_ERR_BAD_FLDNAME)
{
    /*-----*/
    * TIF doesn't know about this field, but maybe ILTR does.  This
    * is how we get info on MAPPED source fields and UNMAPPED
    * target fields.
    /*-----*/
    rc = ILFldTypeEx ( tr, szFieldName, pFieldType, pFieldAttributes,
                      pMaxLength, szTypeDesc );
    if (rc != SUCCESS)
        rc = TIF_ERR_ILFldTypeEx;
}

return rc;
} //---- TIFGetFieldAttributes

/*-----*/
* Name:      ILTIFGetFieldAttributes
* Purpose:   Get parts of a field definition that is stored in the TIF database
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*
* NOTE:     this function tries to be very clever; be sure it isn't too clever
*           for your needs.  It tries to get you attribute info about one of
*           "YOUR" fields, no matter who YOU are.  Depending on what ILTR PHASE
*           we're in, and whether Field Mapping is a factor, we may get the
*           info from TIF itself, or from ILTR.
*
* We always ask TIF first, because it's always possible for translators to
* define fields in TIF that don't exist in the ILTR field list.
/*-----*/
TIF_DLL_ENTRYPOINT ILTIFGetFieldAttributes
(
    ILTR_PTRANSL tr,
    IL_PSTR szFieldName,          // IN: Field Name
    INT32 IL_DIST *pMaxLength,    // OUT: Max Field size (or 0 for unlimited)
    char IL_DIST *pFieldType,     // OUT: FldType (ILX_TYPE_XXX)
    ILTB_ATTRIB IL_DIST *pFieldAttributes ) // OUT: Field Attribute bits
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    int rc;
    rc = TIFGetFieldAttributes ( tr, szFieldName, pMaxLength, pFieldType,
                                pFieldAttributes, NULL );

    return rc;
} //---- ILTIFGetFieldAttributes

/*-----*/
* Name:      TIFGetTargetFieldAttributes
* Purpose:   Get parts of a field definition that is stored in the TIF database
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*
* NOTE:     this function doesn't try to be clever.  Just does what it says.

```

```

/*-----*/
int TIFGetTargetFieldAttributes
(
    ILTR_PTRANS� tr,
    IL_PSTR FieldName,
    INT32 IL_DIST *pMaxLength, // OUT: Max Field size (or 0 for unlimited)
    char IL_DIST *pFieldType, // OUT: FldType (ILX_TYPE_XXX)
    ILTB_ATTRIB IL_DIST *pFieldAttributes ) // OUT: Field Attribute bits
{
    TIF_FIELD_DESC_PTR pDefinition;
    int rc = TIFLookUpFieldDefinition (&ILTIF_pstTIF, FieldName, &pDefinition);
    if (rc == SUCCESS)
    {
        *pMaxLength = pDefinition->MaxLength;
        *pFieldType = pDefinition->FieldType;
        *pFieldAttributes = pDefinition->FieldAttributes;
    }
    return rc;
} //---- TIFGetTargetFieldAttributes

/*-----*/
* Name:      ILTIFFillFields
* Purpose:   Do a standard fill of the TIF mechanism fields, by calling
*            ILTIFDefField once for every field returned by ILGetField
* Input:     tr struct
*            bMappedOnly
*            Should only mapped fields be added
*            bIDNeeded
*            Is an ID needed.
* Return:    SUCCESS if all went well
*            ILTR_ERR_BADMAP field information is inaccessible
*            An error value bubbled up from a subfunction call
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1993
* Note:      Even if all fields are added only mapped fields are involved
*            in reconciliation. If an ID Field is required, then one is
*            added, 32 character text no reconciliation.
/*-----*/
TIF_DLL_ENTRYPOINT ILTIFFillFields ( ILTR_PTRANS� tr,
                                     BOOLEAN bMappedOnly,
                                     BOOLEAN bIDNeeded )
{
    int rc = SUCCESS;
    char szFld[ILTR_MAX_FLDNAME]; // Name of the field
    ILTR_NDX fldnum;
    ILTR_NDX fldmax;
    ILTR_FLDPTR fldlst;
    INT32 ExtraAttributes;

    if (ILTR_direction == ILTR_EXPORT)
    {
        fldmax = ILTR_map.nSource;
        fldlst = ILTR_map.pSource;
    }
    else
    {
        fldmax = ILTR_map.nTarget;
        fldlst = ILTR_map.pTarget;
    }

    // Put out any HIDDEN fields first...
    for (fldnum= 0; fldnum < fldmax && rc == SUCCESS; fldnum++)
    {
        if (fldlst[fldnum].Attribs & ILTB_ATT_HIDDEN_FIELD)
            rc = ILTRTIFDefFieldN (tr, fldnum, 0);
    }

    if (rc != SUCCESS)
        return rc;

    // Now put out non-hidden fields...
    for (fldnum= 0; fldnum < fldmax && rc == SUCCESS; fldnum++)
    {
        //---- skip over any hidden fields on this pass
        if (fldlst[fldnum].Attribs & ILTB_ATT_HIDDEN_FIELD)

```

```

        continue;

        // Get Field Name and Determine whether field is mapped
        int nMapped;
        nMapped = ILFldNameByIndex (tr, fldnum, szFld, sizeof(szFld));
        if (nMapped == FAILURE)
            //--- we come here for sub-items in field list; skip them!
            continue;

        // If field is unmapped and we only want MAPPED fields then skip it
        if (!nMapped && bMappedOnly)
            continue;

        rc = ILTRTIFDefFieldN (tr, fldnum, 0);
    }

    if (rc == SUCCESS)
    {
        ExtraAttributes = TIFEA_IS_MAPPED;
        rc = ILTIFDefFieldEx ( tr, ILTR_SUB_TYPE, 0,
                             "N", // ILX_TYPE_NUMBER
                             NULL,
                             ILTB_ATT_HIDDEN_FIELD | ILTB_ATT_KEY_FIELD,
                             &ExtraAttributes, NULL );
    }

    if (rc == SUCCESS)
        rc = ILTIFStartNextPhase(tr, TIF_PHASE_DONE_SETTING_THINGS_UP);

    return rc;
} //---- ILTIFFillFields

/*-----
* Name:      ILTIFInitRecord
* Create clean slate, ready for 'ILTIFPutField' calls to fill'er up.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFInitRecord ( ILTR_PTRANSL tr )
{
    if (ILTR_pILTIF == NULL )
        return TIF_ERR_PILTIF_IS_NULL;
    else
    {
        PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

        int rc = TIFInitRecord (pstTIF, TIF_pFieldList, &TIF_CurrentRecord);
        if ( (rc == SUCCESS)
            && (TIF_phase == TIF_PHASE_LOADING_SOURCE_RECORDS)
            && (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) )
            // Prime the ILX_V4 Source (cache) Record
            rc = TIFInitRecord (pstTIF, TIF_pSourceFieldList, &TIF_SourceRecord);

        return rc;
    }
} //---- ILTIFInitRecord

/*-----
* ILTIFPutField:
* Place data into a field in the current record. This function assumes
* that character data uses endpoint-specific character encodings. Thus
* all non-binary fields are passed through Character Mapping, which
* includes mapping endpoint-specific EOL character sequences to
* ILTR_EOS_CHAR (0xFF).
*
* For non-binary fields the value supplied as 'lFldSize' helps make
* the debugLog look good, but otherwise it is NOT respected. TIF
* re-computes the Field Length by calling 'strlen'.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFPutField ( ILTR_PTRANSL tr,
                                   IL_PSTR FieldName,
                                   IL_PANY FieldValue,
                                   LONG FieldLength )

```

```

{
    int rc;
    char szTypeDesc[ILTR_MAX_TYPEDESC];
    char fldtype;
    INT32 maxlen;
    ILTB_ATTRIB attribs = 0;
    BOOLEAN bLengthGuaranteed = FALSE;

    //----- Get field attributes
    rc = TIFGetFieldAttributes ( tr, FieldName, &maxlen,
                                &fldtype, &attribs, szTypeDesc );
    if (rc != SUCCESS)
        return ILERROR_S(FieldName, TIF_ERR_ABNORMAL);

    /*-----
    * If this is a TAGGED field, strip tag and put it in _subType field.
    * NOTE: it is illegal for BINARY fields to be TAGGED.
    * NOTE: do nothing if running under a pre-SST stone age app
    *-----*/
    if ((attribs & ILTB_ATT_TAGGED) && ILTR_VERSION_IS_AT_LEAST(21))
    {
        char szTag[ILTR_MAX_TAG_LEN];

        unsigned int len = IL_STRLEN ((IL_PSTR) FieldValue);
        rc = ILSST_StripTag ( tr, (IL_PSTR IL_DIST *) &FieldValue,
                             &len, szTypeDesc, szTag );
        if (rc != SUCCESS)
            return rc;

        if (IL_STRING_ISNT_NULL(szTag))
        {
            rc = ILTIFPutFieldEx2 ( tr, ILTR_SUB_TYPE,
                                    szTag, IL_STRLEN(szTag),
                                    TRUE, FALSE, FALSE );
            if (rc != SUCCESS)
                return ILERROR (rc, TIF_ERR_ABNORMAL);
        }

        FieldLength = (LONG) len;
        bLengthGuaranteed = TRUE;
    }

    /*-----
    * When bLengthGuaranteed=FALSE and fldtype is NOT BINARY, the
    * FieldLength supplied here is eventually supplanted by a
    * computed field length (strlen).
    *-----*/
    rc = ILTIFPutFieldEx2 ( tr, FieldName, FieldValue, FieldLength,
                            bLengthGuaranteed,
                            FALSE, // NOT a special fanning adjustment
                            TRUE ); // YES, do Char Mapping if non-binary

    return rc;
} //----- ILTIFPutField

/*-----
* Name:      ILTIFPutFieldEx
* Extended version of ILTIFPutField, called from ILTR\fldput.c, allows
* for 'SpecialFanningAdjustment' calls and control over Character Mapping.
* Called from ILTR\fldput.c\ILTRPutField, with bMapCharsIfNonBinary=FALSE,
* when char mapping has already been done by the ILTRPutField function.
*
* For non-binary fields the value supplied as 'lFldLen' helps make
* the debugLog look good, but otherwise it is NOT respected. TIF
* re-computes the Field Length by calling 'strlen'.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFPutFieldEx
(
    ILTR_PTRANSL tr,
    IL_PSTR szFldName,
    IL_PANY pFldData,
    LONG lFldLen,
    BOOLEAN bSpecialFanningAdjustment,
    BOOLEAN bMapCharsIfNonBinary )
{

```

```

    return ILTIFFPutFieldEx2 ( tr, szFldName, pFldData, lFldLen,
                               FALSE,    //---- bLengthGuaranteed
                               bSpecialFanningAdjustment,
                               bMapCharsIfNonBinary );
} //---- ILTIFFPutFieldEx

/*-----
* Name:  ILTIFFPutFieldEx2
*
* NOTE:  for BINARY fields, TIF always uses 'lFldSize' w/o question.
*        but for non-BINARY fields, TIF normally does an 'strlen'
*        call to compute the length of the field value.  However if
*        bLengthGuaranteed==TRUE then TIF uses lFldSize as passed in.
*
*        Length supplied by caller must be exact; NOT including any
*        NULL terminator.  (For non-BINARY fields TIF stores LEN+1
*        bytes of data, where the last byte is a NULL supplied by
*        TIF itself.)
*
* This DLL Entrypoint is now obsolete.  ILTIFFN.H no longer lists it.
* But it will be retained for awhile to avoid nuisance revlocks between
* TIF DLL and translators build with ILTR libraries including FLDPUT.OBJ
* from 1/3/96 through 1/19/96.  Eventually this can be made into a static
* (local) function.
*-----*/
extern "C" TIF_DLL_ENTRYPOINT ILTIFFPutFieldEx2
( ILTR_PTRANS tr,
  IL_PSTR szFldName,
  IL_PANY pFldData,
  LONG lFldSize,
  BOOLEAN bLengthGuaranteed,    // see NOTE above
  BOOLEAN bSpecialFanningAdjustment,
  BOOLEAN bMapCharsIfNonBinary )
{
    int rc = SUCCESS;

    // Make sure we have a pointer
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    PSTTIF_TYPE pstTIF; pstTIF = &ILTIF_pstTIF;

    if (bLengthGuaranteed)
        lFldSize |= TIF_LENGTH_GUARANTEED; // 0x40000000

    /*-----
    * Eventually I hope that all TIF users will make direct calls to
    * 'ILTIFStartNextPhase'.  But for older translators we can do the
    * phase transition implicitly.
    *-----*/
    if ( (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
        && (TIF_phase == TIF_PHASE_SETTING_THINGS_UP) )
    {
        rc = TIFStartNextPhase(tr, TIF_PHASE_LOADING_TARGET_RECORDS);
        if (rc != SUCCESS)
            EXIT_WITH_ERROR (rc);
    }

    // If this is the first field for this record
    if ( (bSpecialFanningAdjustment == FALSE)
        && (TIF_bNeedPriming == TRUE) )
    {
        // Prime the Primary Record
        rc = TIFInitRecord (pstTIF, TIF_pFieldList, &TIF_CurrentRecord);
        if ( rc == SUCCESS)
            && (TIF_phase == TIF_PHASE_LOADING_SOURCE_RECORDS)
            && (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) )
            // Prime the ILX_V4 Source (cache) Record
            rc = TIFInitRecord (pstTIF, TIF_pSourceFieldList, &TIF_SourceRecord);

        if (rc != SUCCESS)
            EXIT_WITH_ERROR (rc);
    }
}

```

```

    TIF_bNeedPriming = FALSE;
}

if (ILTR_phase != ILTR_PHASE20)
{
    ILUT_PBUFFER pRecord;

    if ( bSpecialFanningAdjustment
        && (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS) )
        pRecord = &TIF_OriginalRecord;
    else
        pRecord = &TIF_CurrentRecord;

    rc = TIFRecordAddFieldValue
        ( pstTIF,
          TIF_pFieldList,
          szFldName, TIF_NOTSET, // specify field by name
          (IL_PSTR) pFldData,
          lFldSize, // length (sometimes ignored for TEXT fields)
          bMapCharsIfNonBinary,
          pRecord );
}
else
{
    /*-----
    * Caller is trying to load a SOURCE APP field into TIF. First let's
    * see if this is an UNMAPPED source field
    *-----*/
    char szUSFName[ILTR_MAX_FLDNAME+1];
    IL_SPRINTF(szUSFName, TIF_USFN_FORMAT, szFldName);
    TIF_FIELD_DESC_PTR pDefinition;

    rc = TIFLookUpFieldDefinition (pstTIF, szUSFName, &pDefinition);
    if (rc == SUCCESS)
    /*-----
    * Lookup for decorated fieldname succeeded, so we know this is an
    * UNMAPPED SOURCE APP field that TIF has been told about.
    * Put field value into the Current Record Value struct in TIF.
    *-----*/
        rc = TIFRecordAddFieldValue
            ( pstTIF,
              TIF_pFieldList,
              "", pDefinition->FieldNum, // specify field by number
              (IL_PSTR) pFldData,
              lFldSize, // length (sometimes ignored for TEXT fields)
              bMapCharsIfNonBinary,
              &TIF_CurrentRecord );

    else if (rc == TIF_ERR_BAD_FLDNAME)
    /*-----
    * This is a MAPPED SOURCE APP field, or an unmapped field that TIF
    * hasn't been told about. The latter case will lead to ERROR!!
    * We used to tolerate that case, but we don't any longer. (10/28/95)
    *
    * Now try to put it in the SOURCE (cache) RECORD. This attempt
    * will fail if fieldname isn't known to TIF.
    *-----*/
        rc = TIFRecordAddFieldValue
            ( pstTIF,
              TIF_pSourceFieldList,
              szFldName, TIF_NOTSET, // specify field by name
              (IL_PSTR) pFldData,
              lFldSize, // length (sometimes ignored for TEXT fields)
              bMapCharsIfNonBinary,
              &TIF_SourceRecord );

    else
        rc = ILERROR(rc, TIF_ERR_ABNORMAL);
}

Exit:

if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 75))
    TIFLogPutField (tr, szFldName, (UINT8 *) pFldData, lFldSize, rc);

```

```

    return rc;
} //---- ILTIFFPutFieldEx2

/*-----
 * Name:      ILTIFFPutRecord
 * Purpose:   Sometimes Map Fields, and store record in TIF database
 * Input:     tr struct
 * Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFFPutRecord (ILTR_PTRANS tr)
{
    int rc;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    PSTTIF_TYPE pstTIF; pstTIF = &ILTIF_pstTIF;
    TIF_PUT_RECORD_OPTION opt;

    /*-----
     * When sanitizing a DELETE, it's OK to have an "un-primed" record here.
     * We might want to call TIFInitRecord here, but that might blow away
     * some useful field data. Instead we're relying on TIFGetRecord to
     * do the TIFInitRecord call for us. (grep for 'xref001')
     *-----*/
    if ( (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
        && (TIF_CurrentRecordOutcome & ILTIF_OUTCOME_DELETE) )
        TIF_bNeedPriming = FALSE; // **xref001**

    /*-----
     * If the record isn't "primed", we know that no fields have been
     * put into it, so it's just a piece of trash. Don't store it.
     *-----*/
    else if (TIF_bNeedPriming == TRUE)
        return TIF_RECERROR;

    switch (TIF_phase)
    {
        case TIF_PHASE_LOADING_TARGET_RECORDS:

            opt = TIFPRO_ANALYZE_AND_STORE_NEW_RECORD;
            break;

        case TIF_PHASE_LOADING_SOURCE_RECORDS:

            if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
            {
                /*-----
                 * The target translator, operating under ILX_V3, is
                 * loading source app data, which it reads from ILIF
                 * and sanitizes before loading into TIF.
                 *-----*/
                opt = TIFPRO_ANALYZE_AND_STORE_NEW_RECORD;
                break;
            }
            else
            {
                /*-----
                 * The source translator, operating under ILX_V4, is loading
                 * data that is subject to Field Mapping. Values for Unmapped
                 * Source Fields are already in TIF_CurrentRecord, but
                 * values for Mapped Source Fields are in TIF_SourceRecord.
                 * We now do Field Mapping to construct corresponding
                 * Target Field Values, which are put into TIF_CurrentRecord,
                 * then finally we store TIF_CurrentRecord.
                 *-----*/
                rc = TIFPutSourceRecord(tr);
                return rc;
            }
        case TIF_PHASE_SANITIZING_SOURCE_RECORDS:

            /*-----

```

```

        * The target translator has sanitized all the mapped fields, but
        * it's up to us to preserve all the un-mapped fields!!
        *-----*/
rc = TIFCopyUnmappedFields(pstTIF);
if (rc != SUCCESS)
    return rc;

//---- Target Translator is re-loading SOURCE APP data.
if (TIF_RecordHasBeenPutSinceLastGet)
{
    //---- looks like source record is being FANNED into TIF; a
    //---- single source record is begin PUT more than once!!
    //---- so now we have to add a new TIF record.
    opt = TIFPRO_ANALYZE_AND_STORE_NEW_RECORD;
}
else
{
    //---- when sanitizing, handle the first PUT
    //---- by updating the original source record
    TIF_lCurrentRecNum = TIF_lOriginalRecNum;
    opt = TIFPRO_ANALYZE_AND_UPDATE_RECORD;
}
break;

default:

    return TIF_ERR_BAD_PHASE_FOR_PUT_RECORD;
}

rc = TIFPutRecord (tr, opt);

//--- set the 'Put Since Last Get' flag, which we pay attention to
//--- during the 'Sanitizing Source Records' phase of operation
TIF_RecordHasBeenPutSinceLastGet = TRUE;

return rc;
} //---- ILTIFPutRecord

/*-----
* Name:    ILTIFWriteRecord
* Purpose: Same as ILTIFPutRecord
*-----*/
TIF_DLL_ENTRYPOINT ILTIFWriteRecord (ILTR_PTRANSL tr)
{
    int rc = ILTIFPutRecord(tr);
    return rc;
} //---- ILTIFWriteRecord

/*-----
* Name:    ILTIFCloneRecord
* Purpose: Make CurrentRecord be an exact copy of OriginalRecord
*-----*/
TIF_DLL_ENTRYPOINT ILTIFCloneRecord (ILTR_PTRANSL tr)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
    else
    {
        PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;
        int rc = TIFCloneRecord (pstTIF);
        TIF_bNeedPriming = FALSE;
        return rc;
    }
} //---- ILTIFCloneRecord

```



```

/*-----
* Name:      ILTIFA.CPP (and ILTIF.CPP)
* Purpose:   These 2 modules contain almost all of the public entrypoints
*           into the ILTIF mechanism.
*
* Functions (local functions are indented):
*
*           ILTIFStartNextPhase -- tell TIF where you're going
*           AnalyzeAndResolveConflicts -- called by ILTIFEndLoad
*           ILTIFEndLoad -- invokes conflict resolution
*           LogRecord -- write "pretty" stuff to XLATE.LOG
*           ILTIFSetPositionAboveTopRecord
*           ILTIFTopRecord
*           ILTIFNextRecord
*           ILTIFGotoRecord
*           ILTIFRecordNum
*           ILTIFReadRecord
*           ILTIFReadNextRecord
*           ILTIFReadRecordNum
*           ILTIFGetField
*           GetMappedField
*           GetSourceAppField
*           GetFieldGuts
*           ILTIFGetAndCopyField - entrypoint called from ILTR\fldget.c
*           ILTIFSetDataConv - does nothing
*           ILTIFSetReconcile - set reconciliation option
*           ILTIFGetOutcome
*           ILTIFRecordAdded
*           ILTIFRecordChanged
*           ILTIFRecordDeleted
*           ILTIFRecordReplaced
*           CreateOneFigMember
*           ILTIFAcceptOutcome
*           ILTIFRejectOutcome
*           MarkFlag2ForAllFIGMembers
*           MarkFlag2ForAllCIGMembers
*           MappedFieldChanged
*           ILTIFFieldChanged
*           ILTIFDump
*           TIFFreeILTIFBuffers
*           ILTIFClose -- shut down ILTIF
*           CloseFileTemporarily
*           ILTIFCloseFileTemporarily
*           ILTIFCloseFileInitially
*           ILTIFReopenFile
*           ILTIFBeginStatusBar
*           ILTIFBeginStatus
*           ILTIFUpdateStatusBar
*           ILTIFUpdateStatus
*           ILTIFEndStatusBar
*           ILTIFUnloadToILIF
*           ILTIFHowManyRecords
*           TIFPutSourceRecord
*           ILTIFDontSyncByID
*           ILTIFFeatureSet
*           ILTIFItemIsRecurring
*           PutAdjustedDates
*           MarkAllRecurringCigMembers
*           LogFig
*           ILTIFFanItem
*           FanItem
*           FanItemUpdate
*           FanItemCigChop
*           SetCigType
*           ILTIFRemoveRecord
*
*           and STUB versions of the following functions:
*
*           ILCRBegin
*           ILCREnd
*
* Author:   Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*           David Boothby, Copyright (c) IntelliLink Corp., 1995
*-----*/

```

```

#include "iltr.h"

#define EXIT_WITH_ERROR(e) {rc=e; goto Exit;}
#define LOG_ERR_AND_EXIT(e1, e2) EXIT_WITH_ERROR(ILERROR(e1, e2))

static int GetFieldGuts ( ILTR_PTRANSL tr,
                          IL_PSTR szFieldName,
                          int nWhich,
                          BOOLEAN bMapCharsIfNonBinary,
                          LONG *pFieldLength,
                          IL_PANY *pFieldValue );

static int MapCharsFromIL_IfNonBinary ( ILTR_PTRANSL tr,
                                         IL_PSTR FieldName,
                                         ILUT_PBUFFER pField,
                                         INT32 *pFieldLength );

static int MarkFlag2ForAllCIGMembers ( ILDFX_PHNDL phFile, INT32 Item,
                                       INT32 FlagsToSet );

static int FanItem (ILTR_PTRANSL tr, int maxFanCount);

static int FanItemUpdate (PSTTIF_TYPE pstTIF, INT32 *pMaster);

static int FanItemCigChop ( PSTTIF_TYPE pstTIF, INT32 Updater,
                           INT32 Updatee, INT32 Middleman );

static void SetCigType ( ILDFX_PHNDL phFile,
                        INT32 Item,
                        INT32 CigType );

/*-----
 * Name:      ILTIFStartNextPhase
 * Purpose: Signals completion of previous phase, and start of next
 * Author:   David Boothby, Copyright (c) IntelliLink Corporation, 1995
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFStartNextPhase ( ILTR_PTRANSL tr, INT16 phase )
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    if (phase == TIF_PHASE_FANNING_BEFORE_UNLOAD)
    {
        ILTIFlogsz
        ("\r\n ----- 'Fanning Before Unload' starts here\r\n");
        return SUCCESS;
    }

    else if (phase == TIF_PHASE_FINISHED_FANNING_BEFORE_UNLOAD)
    {
        ILTIFlogsz
        ("\r\n ----- 'Fanning Before Unload' completed\r\n");
        return SUCCESS;
    }

    else
    {
        int rc = TIFStartNextPhase(tr, phase);
        return rc;
    }
} //---- ILTIFStartNextPhase

/*-----
 * Name:      AnalyzeAndResolveConflicts
 * Called from: ILTIFEndLoad
 * Author:   David Boothby, Copyright (c) IntelliLink Corporation, 1995
 *-----*/
static int AnalyzeAndResolveConflicts (ILTR_PTRANSL tr)
{
    int rc, rc2;
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

```

```

ILTIFlogsz("AnalyzeAndResolveConflicts");

if (TIF_ReconciliationOption == ILX_OPT_NOTIFY)
{
    // Init conflict resolution (ILCR) subsystem
    rc = ILCRBegin( tr, TIF_DLL_InstanceHandle, FALSE );
    if (rc != SUCCESS)
        return ILERROR(rc, TIF_ERR_INIT_FAILURE);
}

/*-----
* Beyond this point all exit paths must call ILCREnd if Option is NOTIFY.
*-----*/

if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 50))
{
    ILTIFlogsz("\r\n ----- TIF Contents before CAAR:\r\n");
    rc = ILTIFDump(tr);
    if (rc != SUCCESS)
        goto CAAR_EXIT;
}

//----- do Conflict Analysis And Resolution
if (ILTR_nSynchronize)
    rc = TIFSynchronization_CAAR (tr);
else
    rc = TIFSmartMerge_CAAR (tr);

//----- NOTE: one possible error here is ILTR_ERR_CANCEL, if user presses
//----- the CANCEL button during Interactive Conflict Resolution
if (rc == ILTR_ERR_CANCEL)
    goto CAAR_EXIT;

if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 50))
{
    ILTIFlogsz("\r\n ----- TIF Contents after CAAR:\r\n");
    rc2 = ILTIFDump(tr);

    //---- pass back ILTIFDump error if it won't clobber previous error
    if (rc == SUCCESS)
        rc = rc2;
}

CAAR_EXIT:
//-----

if (TIF_ReconciliationOption == ILX_OPT_NOTIFY)
{
    rc2 = ILCREnd (tr, TIF_DLL_InstanceHandle);
    if (rc2 != SUCCESS)
        //---- log error but don't pass it back.
        ILERROR (rc2, 0);
}

ILTIFlogszul("AnalyzeAndResolveConflicts ==> rc=%ld", (UINT32) rc);
return rc;
} //---- AnalyzeAndResolveConflicts

/*-----
* Name:      ILTIFEndLoad
* Purpose:   Signal completed load of TIF and kicks off Conflict Resolution
*
* Return:    SUCCESS          - If all went well.
*           or one of many possible error codes
*
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/
TIF_DLL_ENTRYPOINT ILTIFEndLoad(ILTR_PTRANSL tr)
{
    int rc;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
}

```

```

ILTIFlogsz("ILTIFEndLoad");

if (ILTR_direction == ILTR_EXPORT)
{
    rc = TIFStartNextPhase(tr, TIF_PHASE_UNLOADING_FOR_EXPORT);
    if (rc != SUCCESS) return rc;
}
else
{
    rc = ILTIFStartNextPhase(tr, TIF_PHASE_CONFLICT_RESOLUTION);
    if (rc != SUCCESS) return rc;

    rc = AnalyzeAndResolveConflicts(tr);
    if (rc != SUCCESS) return rc;

    /*-----
    * Start the first UNLOADING phase. This is where the OKToProceed
    * function may be called. rc=ILTR_ERR_CANCEL if user says no.
    *-----*/
    rc = TIFStartNextPhase(tr, TIF_PHASE_UNLOADING_TO_TARGET);
    if (rc != SUCCESS) return rc;
}

// Get the current number of records in ILTIF
LONG lNumOfRecs;
rc = ILTIFHowManyRecords (tr, &lNumOfRecs);
if (rc != SUCCESS)
    return rc;

if (ILTR_direction == ILTR_EXPORT)
    ILSetRecCount (tr, lNumOfRecs);

/*-----
* For Synchronization, put the "Updates To First System:" entry in
* XLATE.LOG. We could suppress this entry if count==0, but that might
* confuse people who are accustomed to seeing it.
*-----*/
if (ILTR_nSynchronize)
    if (ILAppendLog (ILTR_hLog, ILTR_hRes, ILTR_MSG_UPDATE_1ST, NULL, NULL))
        return ILTR_ERR_LOGFILE;

return SUCCESS;
} //---- ILTIFEndLoad

/*-----
* Name:      LogRecord
* Purpose:   imitate logging and record counting done in IMPORT.C of ILTR.
*           Called by ILTIFNextRecord, for IGNOREs, when the
*           "bCurrentlyLoggingAndCountingRecords" flag is set
*           (that is during the first pass of UNLOADING, but not when
*           unloading for EXPORT), and called by ILTIFAcceptOutcome and
*           ILTIFRejectOutcome for other outcomes.
*
* NOTE:      ILTR_lImportRecs is the count that appears in IntelliLink Lite
*           in the summary dialog box at the end of a job.
*
*           ILTR_recNum is the count that is recorded in the xlate.log or
*           il.log file. When we're running in ILX_V3 mode, this count is
*           maintained by ILTR IMPORT.C, but in ILX_V4 mode we do it here.
*-----*/
static int LogRecord (ILTR_PTRANSL tr)
{
    int rc;                // Return code
    int i;                 // Loop counter
    int nResID;            // Resource ID
    char szFldName[ILTR_MAX_FLDNAME]; // Field name
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;
    INT32 outcome = TIF_CurrentRecordOutcome;

    /*--- COMMENTED OUT because we end up with too little information in the log
    /*----- do not log entries for Fanned Instances of Recurring Items
    /* if (TIFX_IS_FANNED_INSTANCE(TIF_hFile, TIF_lCurrentRecNum))

```

```

//    return SUCCESS;

if (outcome == TIF_SKIP_FAIL_RANGE)
{
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_RANGE; // #ImportRecs doesn't count IGNORED items
}
else if (outcome & ILTIF_OUTCOME_OBSOLETE)
{
    /*-----
    * We get here when one SOURCE item obsoletes another.  TARGET items
    * that are obsoleted never bubble up to this point, because the
    * TIFPositionToNextRecord function filters them out.
    *-----*/
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_IGNORE; // #ImportRecs doesn't count OBSOLETE items
}
else if (outcome & ILTIF_OUTCOME_IGNORE)
{
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_IGNORE; // #ImportRecs doesn't count IGNORED items
}
else if (outcome
        & (ILTIF_OUTCOME_LEAVE_ALONE | ILTIF_OUTCOME_LEAVE_DELETED))
{
    /*-----
    * LEAVE_ALONE and LEAVE_DELETED records are TARGET APP records.
    * We don't log them and we don't count them.  Target items that are
    * NOT left alone are a shadowy presence behind the Source items
    * which UPDATE or REPLACE them.
    *-----*/
    return SUCCESS;
}
else if (outcome & ILTIF_OUTCOME_ADD)
{
    ILTR_lImportRecs++;
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_ADD;
}
else if (outcome & ILTIF_OUTCOME_DELETE)
{
    ILTR_lImportRecs++; // not really right to count deletes (?)
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_DELETE;
}
else if (outcome & ILTIF_OUTCOME_UPDATE)
{
    ILTR_lImportRecs++;
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_UPDATE;
}
else if (outcome & ILTIF_OUTCOME_REPLACE)
{
    ILTR_lImportRecs++;
    if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE) ILTR_recNum++;
    nResID = ILTR_MSG_REPLACE;
}
else if (outcome & ILTIF_OUTCOME_FANNED)
{
    // don't increment any counts (???)
    nResID = ILTR_MSG_FAN;
}
else
    //---- something unexpected, such as outcome==0.  May need to change
    //---- this if we change the rules for when 'LogRecord' is called.
    return TIF_ERR_WEIRD_OUTCOME;

IL_PSTR lpszViewField;
rc = TIFGetViewField (&ILTIF_pstTIF, &lpszViewField);
if (rc != SUCCESS)
    return rc;

if (lpszViewField == NULL)
    LoadString( TIF_DLL_InstanceHandle, TIF_STR_UNSPECIFIED,
                ILTR_szRecName, MAX_MSG );

```

```

else
{
    IL_SAFE_STRINGCOPY(ILTR_szRecName, lpszViewField);
    /*-----
    * Truncate the name at end of first line (only relevant to
    * multi-item fields).
    *-----*/
    IL_PSTR lpMatch = IL_STRSTR (ILTR_szRecName, ILTR_szLineTerm);
    if (lpMatch != NULL)
        *lpMatch = 0;

    //----- Truncate the name at EOS char if it exists.
    lpMatch = IL_STRCHR (ILTR_szRecName, ILTR_EOS_CHAR);
    if (lpMatch != NULL)
        *lpMatch = 0;
}

//----- Write out log record.
rc = ILAppendLog (ILTR_hLog, ILTR_hRes, nResID, ILTR_szRecName, NULL);
if (rc != SUCCESS)
    return ILTR_ERR_LOGFILE;

//----- Place field errors in log file.
for (i = 0; i < ILTR_nFldErrorNum; i++)
{
    //----- Convert internal to external field name.
    if (ILFldInToEx ( tr,
                     ILTR_fldError[i].szField,
                     szFldName,
                     ILTR_MAX_FLDNAME ))
        IL_STRCPY (szFldName, ILTR_fldError[i].szField);

    //----- Place next field error in log file.
    if (ILAppendLog ( ILTR_hLog,
                     ILTR_hRes,
                     ILTR_fldError[i].nError,
                     szFldName,
                     NULL) )
        return ILTR_ERR_LOGFILE;
}
return SUCCESS;
} //----- LogRecord

/*-----
* Name:      ILTIFSetPositionAboveTopRecord
* Purpose:   Set Position such that a subsequent ILTIFNextRecord or
*            ILTIFReadNextRecord call will get the FIRST record that
*            pertains to the current UNLOADING PHASE.
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
* Notes:     This function simply sets a global used by ILTIFNextRecord
*            to get the next record in order.
* NOTE:      returns SUCCESS even if there are ZERO records to unload.
*
* Note that TIF automatically calls ILTIFSetPositionAboveTopRecord at the
* beginning of each UNLOADING PHASE.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFSetPositionAboveTopRecord (ILTR_PTRANSL tr)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    ILTIFlogsz("\r\n----- ILTIFSetPositionAboveTopRecord -----");

    ILTIF_pstTIF->CurrentRecordNumber = TIF_POSITION_ABOVE_TOP;
    return SUCCESS;
} //----- ILTIFSetPositionAboveTopRecord

/*-----
* Name:      ILTIFTopRecord
* Purpose:   Goto First record from the TIF.
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994

```

```

* Version 2 Author: David Boothby, 1995
*
* NOTE:  returns TIF_ERR_EOF if there are ZERO records for the current
*        UNLOADING PHASE.
*
* Recommendation:  do NOT use this function.
*                  use ILTIFSetPositionAboveTopRecord instead.
*-----*/
extern "C" TIF_DLL_ENTRYPOINT ILTIFTopRecord (ILTR_PTRANSL tr);
extern "C" TIF_DLL_ENTRYPOINT ILTIFTopRecord (ILTR_PTRANSL tr)
{
    int rc = ILTIFSetPositionAboveTopRecord(tr);
    if (rc == SUCCESS)
        rc = ILTIFNextRecord(tr);  // skip forward and log skipped records...

    return rc;
} //---- ILTIFTopRecord

/*-----
* Name:      ILTIFNextRecord
* Purpose:   Position to next record for current UNLOADING PHASE.
*            and LOG any records that are skipped over (ignored)
*
* Recommendation:  unless you want to step through TIF, looking only at
*                  in-memory Outcome indicators, you should avoid calling
*                  this function directly.  Use ILTIFReadNextRecord instead.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFNextRecord (ILTR_PTRANSL tr)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    PSTTIF_TYPE pstTIF; pstTIF = &ILTIF_pstTIF;

    for (;;)  //----- loop until we find something worth processing...
    {
        //---- don't let Field Errors carry over from one record to the next
        ILTR_nFldErrorNum = 0;

        int rc = TIFPositionToNextRecord (pstTIF);
        if (rc == TIF_SKIP_THIS_RECORD || rc == TIF_SKIP_FAIL_RANGE)
        {
            //--- The unloader is supposed to SKIP this record.
            if (TIF_bCurrentlyLoggingAndCountingRecords == FALSE)
                //--- not logging, so just keep searching...
                continue;

            //----- get this record, just for the sake of logging it
            rc = TIFGetRecord ( pstTIF,
                               ILTIF_pstTIF->CurrentRecordNumber );
            if (rc == SUCCESS)
                rc = TIFGetOutcome ( pstTIF,
                                     TIF_CurrentRecordNumber,
                                     &TIF_CurrentRecordOutcome );
            if (rc == SUCCESS)
                rc = LogRecord(tr);

            if (rc != SUCCESS) return rc;  // abnormal error
        }
        else
            //----- we've hit an unskippable record, or an abnormal error...
            return rc;
    }
} //---- ILTIFNextRecord

/*-----
* Name:      ILTIFGotoRecord
* Purpose:   Goto a specified record in the TIF, verifying that the
*            specified record exists and pertains to the current UNLOADING
*            PHASE.
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994

```

```

* NOTE: Code that uses ILTIF should not make assumptions about the
* numbering of TIF records. Record numbers passed to this
* function should NOT be generated by users of ILTIF; rather
* the numbers should be gotten by calling ILTIFRecordNum. A
* typical usage pattern is to do one pass over the TIF, using the
* [Read]NextRecord to iterate; using ILTIFRecordNum to get
* record numbers, storing them, maybe filtering or sorting them,
* then doing a second pass which iterates through the stored
* record numbers, using ILTIFGotoRecord for positioning.
*
* ILTIF users who don't need to do filtering or sorting probably
* don't need to use this function at all.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFGotoRecord (ILTR_PTRANSL tr, LONG lRecNum)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
    else
    {
        int rc = TIFValidateRecord (&ILTIF_pstTIF, lRecNum);
        if (rc == SUCCESS)
            ILTIF_pstTIF->CurrentRecordNumber = lRecNum;

        ILTIFlogszulul ( "*** ILTIFGotoRecord #%ld ==> rc %ld",
                        (UINT32) lRecNum,
                        (UINT32) rc );

        return rc;
    }
} //---- ILTIFGotoRecord

/*-----
* Name: ILTIFRecordNum
* Purpose: Return the current record number
* Author: Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/
TIF_DLL_ENTRYPOINT ILTIFRecordNum (ILTR_PTRANSL tr, LONG *lRecNum)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    *lRecNum = ILTIF_pstTIF->CurrentRecordNumber;

    return SUCCESS;
} //---- ILTIFRecordNum

/*-----
* Name: ILTIFReadRecord
* Purpose: Read "current" record from the TIF and log it
* Author: Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*
* NOTE: Recommendation: do not use this function.
* use ILTIFReadNextRecord instead.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFReadRecord (ILTR_PTRANSL tr)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    int rc;
    PSTTIF_TYPE pstTIF; pstTIF = &ILTIF_pstTIF;

    /*-----
    * For older code, which may use this function for the first call in
    * an UNLOADING PHASE, we need to position forward before reading.
    * This makes the first ReadRecord call behave like ReadNextRecord.
    *-----*/
    if (TIF_CurrentRecordNumber == TIF_POSITION_ABOVE_TOP)
    {
        rc = ILTIFNextRecord(tr); // skip forward and log any skipped records
        if (rc != SUCCESS)
            return rc;
    }
}

```



```

    }

    rc = TIFGetRecord (pstTIF, TIF_CurrentRecordNumber);
    if (rc != SUCCESS)
        return rc;

    rc = TIFGetOutcome ( pstTIF,
                        TIF_CurrentRecordNumber,
                        &TIF_CurrentRecordOutcome );
    if (rc != SUCCESS)
        return rc;

    if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 60))
    {
        char szBuf[60], szBuf2[100];

        IL_SPRINTF( szBuf, "      Record #%ld [%ld, %ld] was read",
                    TIF_CurrentRecordNumber,
                    TIF_lCurrentRecNum,
                    TIF_lOriginalRecNum );

        if ( (TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET)
            || (TIF_phase == TIF_PHASE_UNLOADING_TO_SOURCE)
            || (TIF_phase == TIF_PHASE_UNLOADING_TO_HISTORY) )
        {
            INT32 Outcome = TIF_CurrentRecordOutcome;
            IL_PSTR lpszOutcome;
            IL_PSTR lpszAck;

            if (Outcome & ILTIF_OUTCOME_LEAVE_DELETED)
                lpszOutcome = "LeaveDeleted";
            else if (Outcome & ILTIF_OUTCOME_LEAVE_ALONE)
                lpszOutcome = "LeaveAlone";
            else if (Outcome & ILTIF_OUTCOME_ADD)
                lpszOutcome = "Add";
            else if (Outcome & ILTIF_OUTCOME_DELETE)
                lpszOutcome = "Delete";
            else if (Outcome & ILTIF_OUTCOME_REPLACE)
                lpszOutcome = "Replace";
            else if (Outcome & ILTIF_OUTCOME_UPDATE)
                lpszOutcome = "Update";
            else if (Outcome & ILTIF_OUTCOME_IGNORE)
                lpszOutcome = "Ignore";
            else if (Outcome & ILTIF_OUTCOME_OBSOLETE)
                lpszOutcome = "Obsoleted";
            else
                lpszOutcome = "???bad???";

            if (Outcome & ILTIF_OUTCOME_DELTA_ACK)
                lpszAck = "/DeltaACK";
            else
                lpszAck = "";

            IL_SPRINTF (szBuf2, "%s, outcome=%s%s", szBuf, lpszOutcome, lpszAck);
            ILTIFlogsz (szBuf2);
        }
        else
            ILTIFlogsz(szBuf);
    }

    //--- clear the 'Put Since Last Get' flag, which we pay attention to
    //--- during the 'Sanitizing Source Records' phase of operation
    TIF_RecordHasBeenPutSinceLastGet = FALSE;

    return rc;
} //---- ILTIFReadRecord

/*-----
* Name:      ILTIFReadNextRecord
* Purpose:   Goto next record from the TIF and read it and log it
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*
* A typical read iteration loop might look like this:

```

```

*
*   ILTIFSetPositionAboveTopRecord(tr)
*   for (;;)
*   {
*       rc = ILTIFReadNextRecord(tr);
*       if (rc != SUCCESS) break;    // could be TIF_ERR_EOF
*       ...use the record...
*   }
*
* Note that TIF automatically calls ILTIFSetPositionAboveTopRecord at the
* beginning of each UNLOADING PHASE.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFReadNextRecord (ILTR_PTRANSL tr)
{
    int rc = ILTIFNextRecord(tr); // skip forward and log skipped records
    if (rc == SUCCESS)
        rc = ILTIFReadRecord(tr);
    return rc;
} //---- ILTIFReadNextRecord

/*-----*/
* Name:      ILTIFReadRecordNum
* Purpose: Goto specified record from the TIF and read it.
* Author: Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/
TIF_DLL_ENTRYPOINT ILTIFReadRecordNum (ILTR_PTRANSL tr, LONG lRecNum)
{
    int rc = ILTIFGotoRecord (tr, lRecNum);
    if (rc == SUCCESS)
        rc = ILTIFReadRecord(tr); // read and maybe log

    return rc;
} //---- ILTIFReadRecordNum

/*-----*/
* Name:      ILTIFGetField
*
* Purpose: Get field value for given field, into a buffer owned by TIF, and
*          return Length == STRLEN(text)+1 for text fields, or "Exact Length"
*          for binary fields.  Field values are NOT truncated.
*
* NOTE: see enormous essay at top of this module for description of the
*        4 distinct scenarios where this function is called.
*
* Inputs:  tr
*          Global tr struct
*          szFldName
*          Field name to get data from
*          nWhich
*          Which data is required:
*              TIF_ORIGINAL - For original data
*              TIF_CURRENT  - For current data
*              TIF_AUTO     - To automatically determine
* Return:  SUCCESS          - If all went well.
*          TIF_ERR_MEM      - If there was a memory problem.
*          lField, hField, and pField are updated
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
TIF_DLL_ENTRYPOINT ILTIFGetField ( ILTR_PTRANSL tr,
                                   IL_PSTR szFieldName,
                                   int nWhich,
                                   LONG *pFieldLength,
                                   IL_PANY *pFieldValue )
{
    int rc;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    rc = GetFieldGuts ( tr, szFieldName, nWhich,
                        TRUE, // Yes, do character mapping for text fields

```

```

        pFieldLength, pFieldValue );
    if (rc != SUCCESS)
        goto Exit;

    /*-----
    * Now we may want to add an SST tag to the data, if all signals are GO.
    *-----*/

    //----- If running under a pre-SST stone age app, do nothing
    if (ILTR_VERSION_IS_PRIOR_TO(21))
        goto Exit;

    //----- If we're NOT importing into a MAIN section, do nothing
    if (ILTR_TargetSST != ILX_SUBSECT_MAIN)
        goto Exit;

    //----- If TAGGING is disabled, do nothing
    if (ILTR_Flags & ILTR_DISABLE_SST_TAGGING)
        goto Exit;

    char szTypeDesc[ILTR_MAX_TYPEDESC];
    char fldtype;
    INT32 maxlen;
    ILTB_ATTRIB attribs;

    //----- Get field attributes
    rc = TIFGetFieldAttributes ( tr, szFieldName, &maxlen,
                                &fldtype, &attribs, szTypeDesc );
    if (rc != SUCCESS)
    {
        rc = ILERROR_S(szFieldName, TIF_ERR_ABNORMAL);
        goto Exit;
    }

    //----- If this isn't a TAGGED field, do nothing
    if ((attribs & ILTB_ATT_TAGGED) == 0)
        goto Exit;

    PSTTIF_TYPE pstTIF; pstTIF = &ILTIF_pstTIF;
    if (TIF_FieldOffset(TIF_pCurrentRecord, TIF_SubTypeFieldNum) == TIF_NOTSET)
    {
        rc = TIF_ERR_NO_SUBTYPE_VALUE;
        goto Exit;
    }

    IL_PSTR pTag;
    pTag = TIF_FieldData(TIF_pCurrentRecord, TIF_SubTypeFieldNum);

    //----- If subtype is zero (ILX_SUBSECT_MAIN) do nothing
    if (IL_STRINGS_EQUAL(pTag, "0"))
        goto Exit;

    rc = ILSST_AddTag ( tr, ILTR_MAX_FIELDLLENGTH+1, pFieldLength,
                        (IL_PSTR IL_DIST *) pFieldValue,
                        pTag, szTypeDesc );

Exit:

    if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 75))
        TIFLogGetField( tr, "ILTIFGetField", szFieldName, nWhich,
                        (UINT8 *) *pFieldValue, *pFieldLength, rc);
    return rc;
} //---- ILTIFGetField

/*-----
* Name:      GetMappedField
* Purpose:   Use ILTR 'ILFldGetEx' function to do Field Mapping to get a
*            field value. Use the ILTIF_Field2 buffer, expanding it as
*            necessary. Return length according to TIF convention: for
*            binary fields return exact length; for non-binary fields
*            return length including trailing NULL.
*
*            Note that ILTIF_Field2 is a reusable buffer, initialized by

```

```

*      TIFInitStruct and freed by ILTIFClose.
*
* Called by GetSourceAppField in Phase40 (unloading into source app).
*      For this call nWhich == TIF_NESTED_CALL and the caller
*      wants us to construct a Source Field Value.
*
* Called by TIFPutSourceRecord in Phase20 (loading from source app)
*      For this call nWhich == TIF_SOURCE_CACHE and the caller
*      wants us to construct a Target Field Value.
*
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
static int GetMappedField ( ILTR_PTRANSL tr,
                           IL_PSTR szFieldName,
                           int nWhich,
                           BOOLEAN bMapCharsIfNonBinary,
                           LONG *pFieldLength,
                           IL_PANY *pFieldValue )
{
    int rc;
    unsigned int uLen;
    char FieldType;
    ILTB_ATTRIB FieldAttributes;
    INT32 MaxLength;

    if (nWhich == TIF_SOURCE_CACHE)

        //---- get field type for SOURCE field
        rc = ILFldTypeInvertedLookup ( tr, szFieldName, &FieldType,
                                       &FieldAttributes,
                                       &MaxLength, NULL );

    else

        //---- get field type for TARGET field
        rc = ILFldTypeEx ( tr, szFieldName, &FieldType, &FieldAttributes,
                           &MaxLength, NULL );

    if (rc != SUCCESS)
        //---- if we can't get field attributes it must be a bad fieldname
        return TIF_ERR_BAD_FLDNAME;

    //----- Get field data, grow field buffer as needed, up to maximum
    for (;;)
    {
        uLen = (unsigned int) (ILTIF_Field2.lBufSize - 2);

        rc = ILFldGetEx ( tr, szFieldName, nWhich, bMapCharsIfNonBinary,
                          (IL_PSTR) ILTIF_Field2.pBuffer, &uLen );
        if (rc != ILTR_ERR_TRUNC)
            break;          // success or non-truncation error

        //---- if buffersize is maxed out, and we still get truncation, give up
        if (ILTIF_Field2.lBufSize == ILTIF_Field2.lMaxSize)
            break;

        /*-----
        * Increase buffer size, but don't exceed maximum.
        * The ILUT Buffer mechanism enforces the maximum buffer size for us.
        *-----*/
        rc = ILUT_GetBuffer (&ILTIF_Field2, ILTIF_Field2.lBufSize + TIF_BUF_INC);
        if ((rc != SUCCESS) && (rc != ILUT_ERR_SETTLE_FOR_LESS))
            return ILERROR(rc, TIF_ERR_MEM);
    }

    switch (rc)
    {
        case SUCCESS:
        case ILTR_ERR_NODATA:
        case ILTR_ERR_NOTMAPPED:
        case ILTR_ERR_NOFLD:      break;    // non-errors

        default:

            ILTIFlogszsul
            ( "ILFldGetEx call, from TIF, for field '%s', got error %ld",

```

```

        szFieldName, (UINT32) rc );

    //---- IGNORE truncation error, but bomb out on serious errors
    if (rc != ILTR_ERR_TRUNC)
        return rc;
}

if (FieldType == ILX_TYPE_BINARY)
    *pFieldLength = (LONG) uLen;
else
    //---- not binary, so increase length to allow for null terminator
    *pFieldLength = (LONG) uLen + 1;

*pFieldValue = ILTIF_Field2.pBuffer;
return SUCCESS;
} //---- GetMappedField

/*-----
 * GetSourceAppField:
 *
 * This function is called by GetFieldGuts when we are unloading
 * TIF data into the SOURCE APP, so we are
 * being asked to supply values for SOURCE APP fields. Since
 * the TIF database consists primarily of TARGET APP fields,
 * we may need to do Field Mapping. But first let's see whether
 * the requested field is one of these weirdo unmapped source fields.
 *
 * We create a decorated unmapped source field name, and try to get it
 * from the ORIGINAL SOURCE record. If that fails with error
 * TIF_ERR_BAD_FLDNAME, then we know that the requested field is NOT
 * an unmapped field that TIF knows about. It is either a mapped
 * field, or an unmapped field that TIF doesn't know about. In either
 * case we then try field mapping...
 *
 * NOTE: this function should always do character mapping.
 *-----*/
static int GetSourceAppField ( ILTR_PTRANSI tr,
                              IL_PSTR szFieldName,
                              LONG *pFieldLength,
                              IL_PANY *pFieldValue )
{
    int rc;

    /*-----
     * First guess it's an unmapped source field, try to get its value.
     *-----*/
    char szUSFName[ILTR_MAX_FLDNAME+1];
    IL_SPRINTF(szUSFName, TIF_USFN_FORMAT, szFieldName);

    rc = TIFGetField ( &ILTIF_pstTIF,
                      szUSFName,
                      TIF_ORIGINAL,
                      pFieldLength,
                      &ILTIF_Field );

    if (rc != TIF_ERR_BAD_FLDNAME)
    {
        //---- either we have succeeded, or we've hit a bad error...
        if (rc == SUCCESS)
            rc = MapCharsFromIL_IfNonBinary ( tr, szFieldName, &ILTIF_Field,
                                              pFieldLength );

        *pFieldValue = ILTIF_Field.pBuffer;
    }
    else
    {
        /*-----
         * It turned out not to be an unmapped source field, so now we
         * try Field Mapping...
         *-----*/
        rc = GetMappedField ( tr, szFieldName, TIF_NESTED_CALL,
                              TRUE, //--- YES, do character mapping
                              pFieldLength, pFieldValue );
    }
}

```

```

    //---- pass back a civilized error code for bad field name
    if (rc == TIF_ERR_BAD_FLDNAME)
        return rc;

    //---- complain bitterly about any other error
    else if (rc != SUCCESS)
        return ILERROR(rc, TIF_ERR_ABNORMAL);
}

/*-----
 * Now it would be nice to honor the 'Value Required' attribute here,
 * but right now we could only do that for UNMAPPED source fields.
 * Eventually we'll build the mechanism to do it for mapped fields too.
 *-----*/
//if ( (rc == SUCCESS)
//    && (*pFieldLength <= 1)
//    && (FieldAttributes & ILTB_ATT_VAL_REQUIRED) )
//{
//    rc = GetDefaultValue();
//}

return rc;
} //---- GetSourceAppField

/*-----
 * GetFieldGuts
 * called by ILTIFGetField and by ILTIFGetAndCopyField
 *-----*/
static int GetFieldGuts ( ILTR_PTRANSI tr,
                        IL_PSTR szFieldName,
                        int nWhich,
                        BOOLEAN bMapCharsIfNonBinary,
                        LONG *pFieldLength,
                        IL_PANY *pFieldValue )
{
    int rc;
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

    /*-----
     * Set default field value -- a zero-length string -- which is
     * what callers expect to see whenever any error is encountered.
     *-----*/
    static IL_PSTR z = "";
    *pFieldValue = z;
    *pFieldLength = 0;

    if ( (ILTR_Flags & ILTR_FLAG_FANNING)
        && ( IL_STRINGS_EQUAL(szFieldName, ILTR_REP_BASIC)
            || IL_STRINGS_EQUAL(szFieldName, ILTR_REP_XDATE) ) )
    {
        /*-----
         * We're in the middle of a FANNING operation, so we want the
         * user to see NULL values for REPEAT pattern and exclusions list.
         *-----*/
        return SUCCESS;
    }

    else if (ILTR_phase == ILTR_PHASE20)
    {
        /*-----
         * We're in the middle of LOADING SOURCE APP data into TIF. The caller
         * specifies a SOURCE APP field name. If the field is mapped, we find
         * the value in the SOURCE CACHE. Otherwise we look for the decorated
         * USFName (Unmapped Source Field Name) in the Target Field List.
         *
         * There are several ways to get here. Calls in iltr\export.c and
         * iltr\sst.c to ILTRGetField end up here. Also when export calls
         * ILTIFPutRecord, that calls TIFPutSourceRecord, which calls
         * GetMappedField for each mapped source field.
         *
         * We never do character mapping here.
         *-----*/
        rc = TIFGetField ( pstTIF,
                        szFieldName,

```

```

        TIF_SOURCE_CACHE,
        pFieldLength,
        &ILTIF_Field3 );

//---- if no such field in source cache, maybe it's an unmapped field
if (rc == TIF_ERR_BAD_FLDNAME)
{
    char szUSFName[ILTR_MAX_FLDNAME+1];
    IL_SPRINTF(szUSFName, TIF_USFN_FORMAT, szFieldName);

    rc = TIFGetField ( pstTIF,
                      szUSFName,
                      TIF_CURRENT,
                      pFieldLength,
                      &ILTIF_Field3 );
}

*pFieldValue = ILTIF_Field3.pBuffer;
return rc;
}
else if ( (ILTR_phase == ILTR_PHASE40)
          && (nWhich != TIF_NESTED_CALL)
          && (nWhich != TIF_FANNING_ADJ) )
{
    /*-----
    * We are unloading TIF data into the SOURCE APP, so we are
    * being asked to supply values for SOURCE APP fields. We may either
    * read an UNMAPPED source value from TIF, or call ILTR ILFldGetEx
    * to invoke Field Mapping, which may result in 1 or more NESTED
    * calls back to ILTIFGetAndCopyField!!!
    *
    * Store "nWhich" for use in nested call...
    *
    * We always do character mapping here.
    *-----*/
    TIF_nWhich = nWhich;
    rc = GetSourceAppField(tr, szFieldName, pFieldLength, pFieldValue);
    return rc;
}
else
{
    /*-----
    * We're in scenario (i) or (iv), as described in the essay at the
    * top of this module. We're very simply looking for a Target App
    * Field, with no possibility of Field Mapping or other weirdness.
    *-----*/
    if (nWhich == TIF_NESTED_CALL)
        /*---- recall original "nWhich" option
        nWhich = TIF_nWhich;

    rc = TIFGetField ( pstTIF,
                      szFieldName,
                      nWhich,
                      pFieldLength,
                      &ILTIF_Field );

    /*-----
    * Now do character mapping if requested and field is non-binary
    *-----*/
    if ((rc == SUCCESS) && bMapCharsIfNonBinary)
        rc = MapCharsFromIL_IfNonBinary ( tr, szFieldName, &ILTIF_Field,
                                           pFieldLength );

    /*-----
    * Always return buffer pointer. We have to allow for callers who
    * disregard a nonzero return code & de-reference the pointer
    *-----*/
    *pFieldValue = ILTIF_Field.pBuffer;
    return rc;
}
} //---- GetFieldGuts

/*-----

```

```

* MapCharsFromIL_IfNonBinary
*-----*/
static int MapCharsFromIL_IfNonBinary ( ILTR_PTRANSL tr,
                                         IL_PSTR fieldName,
                                         ILUT_PBUFFER pField,
                                         INT32 *pFieldLength )
{
    int rc;
    char fieldType;
    ILTB_ATTRIB fieldAttributes;
    INT32 maxLength;
    IL_PSTR lpszLineTerm;

    rc = ILTIFGetFieldAttributes ( tr, fieldName, &maxLength,
                                   &fieldType, &fieldAttributes );
    if (rc != SUCCESS)
        return ILERROR_S (fieldName, TIF_ERR_ABNORMAL);

    if (fieldType == ILX_TYPE_BINARY)
        return SUCCESS;

    //----- Replace IntelliLink EOS chars with app-specific line terminators
    if (fieldAttributes & ILTB_ATT_MULTLINE)
        lpszLineTerm = ILTR_szLineTerm;

    //----- Replace line terminators with spaces in non-multi-line fields.
    else
        lpszLineTerm = ILTR_SPACE_STR;

    /*-----
    * Do character mapping, and put result string in ILTR_pTmpBuf.
    * ILTR_pTmpBuf is managed as a reusable buffer
    * to minimize heap activity
    *-----*/
    rc = ILMapChars ( (IL_PSTR) pField->pBuffer,
                     IL_STRLEN((IL_PSTR) (pField->pBuffer)),
                     ILTR_EOS_STR, // old line terminator ("\xFF")
                     lpszLineTerm, // new line terminator (e.g. "\r\n")
                     ILTR_sImportCharMap.buffer,
                     ILTR_MAX_FIELDLLENGTH,
                     ILTR_pTmpBuf );
    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_ABNORMAL);

    *pFieldLength = IL_STRLEN ((IL_PSTR) (ILTR_pTmpBuf->pBuffer)) + 1;

    /*-----
    * Make sure ILTIF_Field buffer is big enough.
    *-----*/
    rc = ILUT_GetBuffer (pField, *pFieldLength);
    if (rc != SUCCESS)
        return ILERROR_L (*pFieldLength, TIF_ERR_MEM);

    /*-----
    * Copy converted string from ILTR_pTmpBuf back into ILTIF_Field.
    *-----*/
    IL_MEMCPY ( pField->pBuffer,
                ILTR_pTmpBuf->pBuffer,
                (size_t) *pFieldLength);

    return SUCCESS;
} //---- MapCharsFromIL_IfNonBinary

/*-----
* ILTIFGetAndCopyField:
* called by the GetField function in ILTR\fldget.c
*
* Get field value for given field and copy it into the caller's buffer.
* Return Length == STRLEN(text) (NOT STRLEN()+1) for text fields,
* or "Exact Length" for binary fields. Field values ARE truncated as
* necessary, and ILTR_ERR_TRUNC error is returned when that happens.
* (Called from within the ILTR\fldget.c\GetField function.)
*-----*/

```



```

* NOTE: For text, the IN value of *pLength is max size INCLUDING null
*       terminator, but the OUT value of *pLength is STRLEN(text)
*       -- not including null.
*
* So to GetAndCopy the value "1995", you must supply *pLength >= 5,
* and when the call completes you'll have *pLength == 4.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFGetAndCopyField
(
    ILTR_PTRANS tr,
    IL_PSTR FieldName,           // Field Name
    int nWhich,                 // Current or original
    INT32 *pLength,             // IN=MAX / OUT=ACTUAL length of value
    IL_PANY *pBuffer )          // Buffer for the field data
{
    int rc;
    IL_PANY pvData = NULL;
    INT32 FullLength = 0;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    /*-----
    * Get field value. For TEXT, FullLength includes NULL.
    *-----*/
    rc = GetFieldGuts ( tr, FieldName, nWhich,
                        FALSE, // don't do character mapping
                        &FullLength, &pvData);

    if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 98))
        TIFLogGetField( tr, " ILTIFGetAndCopyField", FieldName, nWhich,
                        (UINT8 *) pvData, FullLength, rc );

    if (rc != SUCCESS)
        return rc;

    /*-----
    * For zero-length values take the easy way out...
    *-----*/
    if (FullLength == 0)
    {
        *pLength = 0;
        IL_MAKE_STRING_NULL((IL_PSTR) pBuffer);
        return SUCCESS;
    }

    /*-----
    * Figure out if we need to truncate, and copy what we can
    *-----*/
    size_t RawCopyLength;

    if (FullLength > *pLength)
    {
        RawCopyLength = (size_t) *pLength;
        rc = ILTR_ERR_TRUNC;
    }
    else
    {
        RawCopyLength = (size_t) FullLength;
        rc = SUCCESS;
    }

    IL_MEMCPY (pBuffer, pvData, RawCopyLength);

    /*-----
    * Now we need to know whether the field is binary or not, because
    * that tells us whether to null-terminate and what length to return.
    *-----*/
    INT32 maxlen;
    char FieldType;
    ILTB_ATTRIB FieldAttributes;
    int rc2;

    /*-----
    * For un-nested calls we rely on the standard wisdom to look in the
    * right place for the field type. But when we're handling a nested

```

```

    * call, the field type we're looking for is in the opposite place
    * from where we normally expect to find it...
    *-----*/
    if (nWhich == TIF_NESTED_CALL)
    {
        rc2 = ILFldTypeInvertedLookup ( tr, FieldName,
                                         &FieldType,
                                         &FieldAttributes,
                                         &maxlen, NULL );

        if (rc2 != SUCCESS)
            rc2 = TIF_ERR_ILFldTypeInvertedLookup;
    }
    else if (nWhich == TIF_FANNING_ADJ)
        rc2 = TIFGetTargetFieldAttributes ( tr, FieldName, &maxlen, &FieldType,
                                             &FieldAttributes );
    else
        rc2 = ILTIFGetFieldAttributes ( tr, FieldName, &maxlen, &FieldType,
                                         &FieldAttributes );
    if (rc2 != SUCCESS)
    {
        /*-----*/
        * Inability to get field attributes is an error which we want to
        * complain about, but just in case caller ignores return code we
        * make the most conservative assumption (to set up length and
        * value assuming the field contains null-terminated text).
        *-----*/
        FieldType = ILX_TYPE_TEXT;
        rc = rc2; // this error masks any previous error
    }

    if (FieldType == ILX_TYPE_BINARY)
        /*--- for binary fields, simply return exact length
        *pLength = RawCopyLength;
    else
    {
        /*-----*/
        * Do null-termination and return length for text
        *-----*/
        *pLength = RawCopyLength - 1;
        ((IL_PSTR) pBuffer) [RawCopyLength-1] = 0;
    }

    return rc;
} //---- ILTIFGetAndCopyField

/*-----*/
* Name:      ILTIFSetDataConv
* Purpose: Set the data translation option (NO-OP; unused!!)
*-----*/
TIF_DLL_ENTRYPOINT ILTIFSetDataConv (ILTR_PTRANSL tr, int nOption)
{
    return SUCCESS;
} //---- ILTIFSetDataConv

/*-----*/
* Name:      ILTIFSetReconcile
* Purpose: Set the reconciliation option for TIF mechanism
*
* NOTE: this is an archaic function, but it's still used.
*       The old guts of this function have been moved into
*       the 'AnalyzeAndResolveConflicts' function.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFSetReconcile (ILTR_PTRANSL tr, int nOption)
{
    int rc = SUCCESS;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    ILLOG_logszul ( ILTIFLOG, ILLOG_TINY_STEP,
                   "ILTIFSetReconcile %ld", (UINT32) nOption );

```

```

/*-----
 * Eventually I hope that all TIF users will make direct calls to
 * 'ILTIFStartNextPhase'. But for older translators we can do the
 * phase transition implicitly.
 *-----*/
if ( (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
    && (ILTR_direction == ILTR_IMPORT) )
{
    rc = TIFStartNextPhase(tr, TIF_PHASE_LOADING_SOURCE_RECORDS);
    if (rc != SUCCESS) return rc;
}

return rc;
} //---- ILTIFSetReconcile

/*-----
 * Record Outcome Functions:
 *
 * These functions are used, during an UNLOADING PHASE, to get the
 * OUTCOME flags for a given record. The 'GetOutcome'
 * function returns a long word full of flag bits; the other
 * functions in this group call 'GetOutcome', then check
 * for various bit flags in the flag word.
 *
 * Functions are:
 *
 *     ILTIFGetOutcome
 *     ILTIFRecordAdded
 *     ILTIFRecordChanged
 *     ILTIFRecordDeleted
 *     ILTIFRecordReplaced
 *
 *     ILTIFFieldChanged
 *
 * All of the Record-level functions work from in-memory TIF
 * knowledge, so they need not be preceded by a ReadRecord call.
 *
 * Unlike the Record-level functions, the ILTIFFieldChanged function
 * must be preceded by a ReadRecord call.
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFGetOutcome
( ILTR_PTRANSL tr, INT32 IL_DIST *pOutcome )
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
    else
    {
        int rc = TIFGetOutcome (&ILTIF_pstTIF,
                                ILTIF_pstTIF->CurrentRecordNumber,
                                pOutcome);

        if (rc != SUCCESS)
            return rc;
        else if (*pOutcome & ILTIF_EXPECTED_OUTCOMES)
            return SUCCESS;
        else
            return TIF_ERR_WEIRD_OUTCOME;
    }
} //---- ILTIFGetOutcome

TIF_DLL_ENTRYPOINT ILTIFRecordAdded ( ILTR_PTRANSL tr )
{
    INT32 Outcome;
    int rc = ILTIFGetOutcome(tr, &Outcome);
    if (rc != SUCCESS)
        return rc;

    return ((Outcome & ILTIF_OUTCOME_ADD) != 0);
} //---- ILTIFRecordAdded

```

```

TIF_DLL_ENTRYPOINT ILTIFRecordChanged ( ILTR_PTRANSL tr )
{
    INT32 Outcome;
    int rc = ILTIFGetOutcome(tr, &Outcome);
    if (rc != SUCCESS)
        return rc;

    return ((Outcome & ILTIF_OUTCOME_UPDATE) != 0);
} //---- ILTIFRecordChanged

TIF_DLL_ENTRYPOINT ILTIFRecordDeleted ( ILTR_PTRANSL tr )
{
    INT32 Outcome;
    int rc = ILTIFGetOutcome(tr, &Outcome);
    if (rc != SUCCESS)
        return rc;

    return ((Outcome & ILTIF_OUTCOME_DELETE) != 0);
} //---- ILTIFRecordDeleted

TIF_DLL_ENTRYPOINT ILTIFRecordReplaced ( ILTR_PTRANSL tr )
{
    INT32 Outcome;
    int rc = ILTIFGetOutcome(tr, &Outcome);
    if (rc != SUCCESS)
        return rc;

    return ((Outcome & ILTIF_OUTCOME_REPLACE) != 0);
} //---- ILTIFRecordReplaced

/*-----
 * CreateOneFigMember
 *
 * Used while Fanning During Unload.
 * It is assumed that the TIF_CurrentRecord buffer contains the
 * Recurrence Master Item with all adjustable date fields suitably adjusted.
 *-----*/
static int CreateOneFigMember ( PSTTIF_TYPE pstTIF,
                               ILUT_PBUFFER pRecBuf,
                               INT32 IL_DIST *pRecNum)
{
    int rc;
    int FieldCount;
    int fldnum;
    int OtherIDFldNum;
    INT32 recnum;
    INT32 FannedForWhom;
    INT32 InstanceFlags;
    TIF_RECORD_VALUE_PTR pRec;

    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 Master = TIF_lCurrentRecNum;

    //---- Determine what flag bits to set for the Fanned Instance Items.
    if (TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET)
    {
        FannedForWhom = TIF_IS_FANNED_FOR_TARGET;
        OtherIDFldNum = TIF_SourceIDFieldNum;
    }
    else if (TIF_phase == TIF_PHASE_UNLOADING_TO_SOURCE)
    {
        FannedForWhom = TIF_IS_FANNED_FOR_SOURCE;
        OtherIDFldNum = TIF_TargetIDFieldNum;
    }
    else
        return ILERROR(TIF_phase, TIF_ERR_BAD_STATE);

    //--- set origin of instances to be same as origin of master

```

```

InstanceFlags = FannedForWhom | TIFX_ORIGIN(phFile, Master);

/*-----
 * Build Instance in buffer pointed to by pRecBuf
 *-----*/
rc = TIFInitRecord (pstTIF, TIF_pFieldList, pRecBuf);
if (rc != SUCCESS)
    LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

/*-----
 * Copy all non-repeat info from master into instance
 *-----*/
FieldCount = TIF_FieldCount(pstTIF);
for (fldnum = 0; fldnum < FieldCount; fldnum++)
{
    /*-----
     * Don't put REPEAT info into fanned instances, and don't copy
     * "other ID" into fanned instances.
     *-----*/
    if ( (fldnum == TIF_RepBasicFieldNum)
        || (fldnum == TIF_RepExclFieldNum)
        || (fldnum == OtherIDFldNum) )
        continue;

    if (TIF_FieldOffset(TIF_pCurrentRecord, fldnum) != TIF_NOTSET)
    {
        INT32 len = TIF_FieldLength(TIF_pCurrentRecord, fldnum);
        IL_HPSTR p = TIF_FieldData(TIF_pCurrentRecord, fldnum);
        if (len > 0)
        {
            rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList,
                                         "", fldnum,
                                         p, len,
                                         FALSE, // do no character mapping
                                         pRecBuf );

            if (rc != SUCCESS)
                LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);
        }
    }
}

// Increment the number of records in the file
recnum = TIF_TotalRecordCount++;

INT32 exdata[TIF_EXDATA_PER_RECORD];
IL_MEMSET((IL_PANY) exdata, 0, sizeof(exdata));

exdata[TIF_FLAGS_SLOT] = InstanceFlags;
exdata[TIF_NEXT_IN_CIG_SLOT] = recnum; // singleton CIG
exdata[TIF_NEXT_IN_SKG_SLOT] = recnum; // singleton SKG
exdata[TIF_NEXT_IN_FIG_SLOT] = recnum; // singleton FIG

pRec = (TIF_RECORD_VALUE_PTR) pRecBuf->pBuffer;

//---- compute hash values and start/end DTTM values
rc = TIFComputeSearchKeyValues(pstTIF, pRec, exdata);
if (rc != SUCCESS)
    LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

//---- store the new record in the TIF (ILDFX) file
rc = ILDFX_AddRecord (phFile, recnum, pRec, TIFREC_SIZE(pRec), exdata);
if (rc != SUCCESS)
    LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

TIFX_NEXT_IN_FIG(phFile, recnum) = TIFX_NEXT_IN_FIG(phFile, Master);
TIFX_NEXT_IN_FIG(phFile, Master) = recnum;

/*-----
 * Mark the Master as Fanned for Source or Fanned for Target.
 * (this is done to the master redundantly, once per instance)
 *-----*/
TIFX_FLAGS(phFile, Master) |= FannedForWhom;

//---- push goalpost out for next unload scan but not for this scan
TIF_GoalPostForNextPass = recnum;

```

```

//---- let caller know record number of newly-created instance record
*pRecNum = recnum;

Exit:

TIFlogszulul ("CreateOneFigMember (item #%ld) rc=%ld",
              (UINT32) recnum, (UINT32) rc );
return rc;
} //---- CreateOneFigMember

/*-----
 * Functions for Accepting or Rejecting a Record Outcome
 *
 * When a translator is unloading records from TIF, TIF assumes that
 * the prescribed record outcomes will be effected by the translator.
 *
 * As long as this assumption holds true, nothing special need be
 * done. But it is a good practice, especially when doing
 * Synchronization, for the translator to call ILTIFAcceptOutcome
 * for each record. Currently this is REQUIRED for INSERT operations
 * done during synchronization if the translator wants to supply a
 * unique ID for the newly-created item, and for UPDATE operations
 * that cause a new unique ID to be assigned by the application.
 *
 * NOTE that TIF does NOT allow for the possibility that a LEAVE_ALONE
 * outcome might result in assignment of a new unique ID. If any
 * application has that behavior then unique IDs are UNUSABLE for
 * IntelliLink synchronization, and the translator must not tell TIF
 * that Unique IDs exist!!
 *
 * TIF allows for the possibility that a sync-driven UPDATE may force
 * assignment of a new ID. However IntelliLink Synchronization can NOT
 * make use of unique IDs that may be altered by any activity other than
 * IntelliLink Synchronization itself.
 *
 * On the other hand if a translator finds that it cannot put a record
 * outcome into effect (e.g. it cannot add a record), it has two
 * choices:
 *
 * 1. it can treat the failure as a FATAL error which aborts the
 *    entire operation, or
 *
 * 2. it can simply tell TIF that the outcome for this particular
 *    record is rejected, then continue processing.
 *
 * For option #2 it must call ILTIFRejectOutcome. At this point in
 * time no values are defined for the 2nd argument to this function.
 */
/*-----
 * ILTIFAcceptOutcome
 * -- for 2nd arg, pass NULL (not "") if you don't have a New Unique ID
 *    to tell TIF about.
 */
TIF_DLL_ENTRYPOINT ILTIFAcceptOutcome (ILTR_PTRANSL tr, IL_PSTR newUniqueID)
{
    int rc = SUCCESS;
    ILUT_PBUFFER pRecBuf;
    TIF_RECORD_VALUE_PTR pRec;
    IL_PSTR fieldName;
    PSTTIF_TYPE pstTIF;
    ILDFX_PHNDL phFile;
    INT32 idHash;
    int idHashSlot;
    INT32 recnum;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    rc = LogRecord (tr);
    if (rc != SUCCESS)
        EXIT_WITH_ERROR (rc);

```

```

if ( (ILTR_nSynchronize == ILXTR_SYNC_NO)
    || (newUniqueID == NULL)
    || IL_STRING_IS_NULL(newUniqueID) )
    EXIT_WITH_ERROR (SUCCESS);

pstTIF = &ILTIF_pstTIF;
phFile = TIF_hFile;
idHash = TIFSimpleHash (&ILTIF_pstTIF, newUniqueID);
idHashSlot = TIF_NOTSET;

if (ILTR_phase == ILTR_PHASE30)
{
    //---- Target App translator is unloading; store Target ID
    fieldName = TIF_TARGETID_FIELDNAME;

    //---- If Target IDs are to be used in Synchronization, compute hash
    if (TIF_bSyncUsingTargetIDs)
        idHashSlot = TIF_TARGETID_HASH_SLOT;

    TIFX_TARGETID_HASH(TIF_hFile, TIF_lCurrentRecNum) = idHash;
}
else if (ILTR_phase == ILTR_PHASE40)
{
    //---- Source App translator is unloading; store Source ID
    fieldName = TIF_SOURCEID_FIELDNAME;

    //---- If Source IDs are to be used in Synchronization, compute hash
    if (TIF_bSyncUsingSourceIDs)
        idHashSlot = TIF_SOURCEID_HASH_SLOT;

    TIFX_SOURCEID_HASH(TIF_hFile, TIF_lCurrentRecNum) = idHash;
}
else
    EXIT_WITH_ERROR (TIF_ERR_BAD_PHASE_FOR_ACCEPT_OUTCOME);

/*-----
 * 2/22/96: NOTE: see version 120 or earlier of this module for a
 * WARNING about new unique IDs assigned by UPDATE. See also the notes
 * for TIFTableAutomatic in tiftable.cpp. At this point I believe that
 * handling of new unique IDs, assigned by either ADD or UPDATE, is
 * fully functional and bug-free. Only likely exception would be for
 * ID-bearing FIG members.
 *-----*/
if (ILTR_Flags & ILTR_FLAG_FANNING)
{
    /*-----
     * Create an instance of the recurring master and link it into
     * the FIG owned by the recurring master. The instance is
     * created in the TIF_FirstRecord buffer and is stored on disk.
     *
     * recnum is then set to identify the newly created instance record
     *-----*/
    pRecBuf = &TIF_FirstRecord;
    rc = CreateOneFigMember (pstTIF, pRecBuf, &recnum);

    /*-----
     * Put special value IDHASH value into Master item to indicate that
     * an ID-bearing FIG is attached to it. This is done redundantly
     * to the Master, once per Instance.
     *-----*/
    if ((rc == SUCCESS) && (idHashSlot != TIF_NOTSET))
        TIFX(phFile, TIF_lCurrentRecNum, idHashSlot) = TIFHASH_SPECIAL;
}
else
{
    //-- current record is the one we'll update and store idHash (if any) in
    recnum = TIF_lCurrentRecNum;
    pRecBuf = &TIF_CurrentRecord;
}

//---- write new unique id into Current TIF record
rc = TIFRecordAddFieldValue ( pstTIF,
                             TIF_pFieldList,
                             fieldName, TIF_NOTSET,

```

```

        newUniqueID, 0,
        FALSE, // don't do character mapping
        pRecBuf );

    if (rc != SUCCESS)
        EXIT_WITH_ERROR (rc);

    //---- update current record on disk
    pRec = (TIF_RECORD_VALUE_PTR) pRecBuf->pBuffer;

    rc = ILDFX_UpdateRecord ( TIF_hFile, recnum,
                             pRec, TIFREC_SIZE(pRec),
                             NULL ); // don't update EXDATA

    if ((rc == SUCCESS) && (idHashSlot != TIF_NOTSET))
        TIFX(phFile, recnum, idHashSlot) = idHash;

Exit:

    ILTIFlogszsul ( "AcceptOutcome(id=%s) ==> rc=%ld",
                    newUniqueID, (UINT32) rc ); // ok to log NULL
    return rc;
} //---- ILTIFAcceptOutcome

/*-----
 * ILTIFRejectOutcome
 *
 * WARNING:  this function is NOT completely implemented for .
 *           Synchronization, but it should work OK for SmartMerge.
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFRejectOutcome (ILTR_PTRANS tr, int ec)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
    else
    {
        ILTIFlogszsul("RejectOutcome, ec=%ld", (UINT32) ec);

        PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

        if ( (TIF_CurrentRecordOutcome == ILTIF_OUTCOME_LEAVE_ALONE)
            && (ec >= 0)
            && (ec <= ILTR_SKIP_ALL) )
            ; // leave error-free "leave_alone" outcomes alone.
        else
            TIF_CurrentRecordOutcome = ILTIF_OUTCOME_IGNORE;

        int rc = LogRecord (tr);
        INT32 Flag2;

        //---- for synchronization, mark all frustrated CIG members
        if (ILTR_phase == ILTR_PHASE30)
            //---- Target App translator is unloading...
            Flag2 = TIF2_TARGET_REJECT;

        else if (ILTR_phase == ILTR_PHASE40)
            //---- Target App translator is unloading...
            Flag2 = TIF2_SOURCE_REJECT;

        else
            return rc;

        rc = MarkFlag2ForAllCIGMembers ( TIF_hFile,
                                         TIF_lCurrentRecNum,
                                         Flag2 );

        return rc;
    }
} //---- ILTIFRejectOutcome

/*-----
 * Name:      MarkFlag2ForAllFigMembers
 * Called by MarkFlag2ForAllCIGMembers

```



```

* Purpose: Set Flag2 Bits on item and any FIG members
*           that are attached to it.
*-----*/
static int MarkFlag2ForAllFigMembers ( ILDFX_PHNDL phFile, INT32 Item,
                                      INT32 FlagsToSet )
{
    INT32 Next = Item;
    int i;

    for (i=1; i <= TIF_MAX_FIG_SIZE; i++) // infinite loop protection
    {
        TIFX_FLAGS2(phFile, Next) |= FlagsToSet;
        Next = TIFX_NEXT_IN_FIG(phFile, Next);
        if (Next == Item)
            break;
    }

    if (i > TIF_MAX_FIG_SIZE)
        return ILERROR_L(Item, TIF_ERR_BROKEN_FIG); // too big or not circular

    return SUCCESS;
} //---- MarkFlag2ForAllFigMembers

/*-----
* Name:      MarkFlag2ForAllCIGMembers
* Purpose: set some bits for all CIG members and any attached FIG members
* Called by ILTIFRejectOutcome to mark rejects
*-----*/
static int MarkFlag2ForAllCIGMembers ( ILDFX_PHNDL phFile,
                                      INT32 Item,
                                      INT32 FlagsToSet )
{
    INT32 Next = Item;
    int i;

    for (i=1; i <= TIF_MAX_CIG_SIZE; i++) // infinite loop protection
    {
        int rc = MarkFlag2ForAllFigMembers (phFile, Next, FlagsToSet);
        if (rc != SUCCESS)
            return rc;

        Next = TIFX_NEXT_IN_CIG(phFile, Next);
        if (Next == Item)
            break;
    }

    if (i > TIF_MAX_CIG_SIZE)
        return ILERROR_L(Item, TIF_ERR_BROKEN_CIG); // too big or not circular

    return SUCCESS;
} //---- MarkFlag2ForAllCIGMembers

/*-----
* Name:      MappedFieldChanged
* Purpose: internal function to help ILTIFFieldChanged do its job
*
* Explanation: when TIF is asked whether a MAPPED field has changed, we
*               can't give a quick & easy answer, due to the vagaries of
*               field mapping. The only reliable way to answer the question
*               is to get the field values and compare them.
*
* Returns TRUE, FALSE, or various error codes...
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
static int MappedFieldChanged (ILTR_PTRANSL tr, IL_PSTR FieldName)
{
    int rc;

    /*-----
    * Get field type, needed for comparing field values
    *-----*/

```

```

    INT32 maxlen;
    char FieldType;
    ILTB_ATTRIB FieldAttributes;
    rc = ILTIFGetFieldAttributes ( tr, FieldName, &maxlen, &FieldType,
                                   &FieldAttributes );

    if (rc != SUCCESS)
        return rc;

    /*-----
    * Get current value of field.
    *-----*/
    INT32 CurrentLength;
    IL_PSTR CurrentValue;
    rc = ILTIFGetField ( tr, FieldName, TIF_CURRENT,
                        &CurrentLength, (IL_PANY *) &CurrentValue);
    if (rc != SUCCESS)
        return rc;

    /*-----
    * Copy current value into a ILTIF_Field3 buffer
    * so it won't be clobbered by next ILTIFGetField call.
    *-----*/
    if (CurrentLength > 0)
    {
        if (CurrentLength > 32767)
            return TIF_ERR_FIELD_VALUE_TOO_BIG;

        /*--- Make sure buffer is big enough; if not then make it bigger
        rc = ILUT_GetBuffer (&ILTIF_Field3, CurrentLength);
        if (rc != SUCCESS)
            return ILERROR_L (CurrentLength, TIF_ERR_MEM);

        /*--- copy field value. We rely on the fact that ILTIFGetField
        /*--- sets CurrentLength = STRLEN + 1 for text fields
        IL_MEMCPY(ILTIF_Field3.pBuffer, CurrentValue, (UINT) CurrentLength);
        CurrentValue = (IL_PSTR) ILTIF_Field3.pBuffer;
        }

    /*-----
    * Get original value of field.
    *-----*/
    INT32 OriginalLength;
    IL_PSTR OriginalValue;
    rc = ILTIFGetField ( tr, FieldName, TIF_ORIGINAL,
                        &OriginalLength, (IL_PANY *) &OriginalValue);
    if (rc == SUCCESS)
    {
        /*-----
        * Compare original value vs current value.
        *-----*/
        BOOLEAN ValuesDiffer;
        ValuesDiffer = TIFFFieldValuesDiffer2( tr, OriginalValue,
                                                CurrentValue,
                                                OriginalLength,
                                                CurrentLength,
                                                FieldType, FieldAttributes );

        rc = (int) ValuesDiffer;
    }

    return rc;
} //---- MappedFieldChanged

/*-----
* Name:      ILTIFFieldChanged
* Purpose:   Has the designated field been altered in the current TIF mechanism
*            record.
* Inputs:    tr
*            Global tr struct
* Return:    TRUE - If it has been altered
*            FALSE - If it has not been altered
*            TIF_ERR_BAD_FLDNAME - If supplied fldname not recognized.
*            ILTR_ERR_BAD_STATE - If there is no current record
*            TIF_ERR_MEM - If there was a memory problem.

```

```

* Author: Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFieldChanged (ILTR_PTRANSL tr, IL_PSTR szFldName)
{
    int rc;

    if (ILTR_pILTIF == NULL)
        rc = TIF_ERR_PILTIF_IS_NULL;
    else if (ILTR_phase == ILTR_PHASE40)
        rc = MappedFieldChanged (tr, szFldName);
    else
        rc = TIFFFieldChanged (&ILTIF_pstTIF, szFldName);
    return rc;
} //---- ILTIFFieldChanged

/*-----
* Name:      ILTIFFDump
* For Debugging, create formatted dump of
* the CURRENTLY OPEN TIF file
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFDump (ILTR_PTRANSL tr)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
    else
        return TIFDump (&ILTIF_pstTIF);
} //---- ILTIFFDump

/*-----
* TIFFFreeILTIFBuffers
*-----*/
void TIFFFreeILTIFBuffers (ILTR_PTRANSL tr)
{
    // Free the Field Buffers if allocated
    ILUT_FreeBuffer (&ILTIF_Field);
    ILUT_FreeBuffer (&ILTIF_Field2);
    ILUT_FreeBuffer (&ILTIF_Field3);
    ILUT_FreeBuffer (&ILTIF_TmpBuf);

    // Free the top-level ILTIF structure
    IL_FREE (ILTIF_hstILTIF, ILTR_pILTIF);
    ILTR_pILTIF = NULL;
    //--- note: don't zero hstILTIF cuz it lives inside *ILTR_pILTIF
} //---- TIFFFreeILTIFBuffers

/*-----
* Name:      ILTIFFClose
* Purpose: Close the TIF mechanism and DELETE the TIF FILE
* Inputs:   tr
*           Global tr struct
* Return:   SUCCESS - If all went well.
*           or one of many possible error codes
*
* Author: Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFClose ( ILTR_PTRANSL tr )
{
    int rc;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    rc = TIFFTerminate (&ILTIF_pstTIF, tr);

    TIFFFreeILTIFBuffers (tr);

    return rc;
} //---- ILTIFFClose

```

```

/*-----
* Name:      CloseFileTemporarily
* Purpose:   Close the TIF workfile file at translation phase transition
* Callers:   ILTIFCloseFileTemporarily and ILTIFCloseFileInitially
*-----*/
static int CloseFileTemporarily ( ILTR_PTRANSL tr,
                                  BOOLEAN bExitPhase,
                                  BOOLEAN bMaybeSaveMaxima )
{
    int rc = SUCCESS;
    int rc2;
    PSTTIF_TYPE pstTIF;
    ILDFX_PHNDL phFile;

    if (ILTR_pILTIF == NULL || ILTIF_pstTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    pstTIF = &ILTIF_pstTIF;
    phFile = TIF_hFile;

    /*-----
    * Save Fanout Maxima if required (do it now, before it's too late!!)
    *-----*/
    if (bMaybeSaveMaxima && TIF_bPleaseSaveFanoutMaxima)
    {
        TIF_bPleaseSaveFanoutMaxima = FALSE;
        rc = TIFSaveFanoutMaxima (tr);
        if (rc != SUCCESS)
            return rc;
    }

    /*-----
    * For unmixed Windows jobs we do nothing at all!
    * For Macintosh we close the workfile and the logfile but in-memory
    * structures are preserved.
    *-----*/
    if (ILTR_Flags & ILTR_FLAG_MACINTOSH)
    {
        TIFlogsz("CloseFileTemporarily/Mac");
        rc = ILDFX_CloseFileTemporarily (phFile);
        rc2 = ILLOG_close (TIFLOG);
        TIF_bWorkFileIsOpen = FALSE;
        return rc;
    }

    /*-----
    * For a MIXED Windows job, where a 32-bit engine is driving one or more
    * 16-bit translators, flush all updates to disk and close the file.
    * When doing this for a 16-bit translator free TIF buffers too.
    *-----*/
    else if (ILTR_Flags & ILTR_FLAG_MIXED_WIN3216)
    {
        /*---- force EXIT from Current Phase, if requested
        if (bExitPhase)
            rc = TIFStartNextPhase (tr, TIF_PHASE_NEXT);

        /*---- close the workfile, flush all updates out to disk
        phFile->bHasChanged = TRUE;
        rc2 = ILDFX_CloseFile (phFile, ILDFX_DO_UPDATE);
        if (rc == SUCCESS)
            rc = rc2;

        #ifdef ILX32_16
        /*---- come here when called by 16-bit translator; do full cleanup
        TIFlogsz("CloseFileTemporarily/WinMix16");
        rc2 = ILLOG_end (&TIFLOG);
        TIFFreeBuffers (tr, pstTIF);
        IL_FREE (TIF_hstTIF, *pstTIF);
        TIFFreeILTIFBuffers (tr);
        #else
        /*---- come here when called by 32-bit translator or engine
        TIFlogsz("CloseFileTemporarily/WinMix32");

```

```

        rc2 = ILLOG_close (TIFLOG);
        TIF_bWorkFileIsOpen = FALSE;
    #endif
}
else
    TIFlogsz("CloseFileTemporarily/NO-OP");

    return rc;
} //---- CloseFileTemporarily

/*-----
* Name:      ILTIFCloseFileTemporarily
* Purpose:   Close the TIF workfile at end of translation phase
* Callers:   ILTR\IMPORT.C, ILTR\EXPORT.C
*-----*/
TIF_DLL_ENTRYPOINT ILTIFCloseFileTemporarily (ILTR_PTRANSL tr)
{
    BOOLEAN bExitPhase = TRUE;
    BOOLEAN bMaybeSaveMaxima = TRUE;
    int rc = CloseFileTemporarily (tr, bExitPhase, bMaybeSaveMaxima);
    return rc;
}

/*-----
* Name:      ILTIFCloseFileInitially
* Purpose:   Close the TIF workfile at end of translation phase
* Callers:   ILX_V3\XLATE.C
*-----*/
TIF_DLL_ENTRYPOINT ILTIFCloseFileInitially (ILTR_PTRANSL tr)
{
    BOOLEAN bExitPhase = FALSE;
    BOOLEAN bMaybeSaveMaxima = FALSE;
    int rc = CloseFileTemporarily (tr, bExitPhase, bMaybeSaveMaxima);
    return rc;
}

/*-----
* Name:      ILTIFReopenFile
* Purpose:   Re-open a TIF file that was previously closed by calling
*            ILTIFCloseFileTemporarily or ILTIFCloseFileInitially
*-----*/
TIF_DLL_ENTRYPOINT ILTIFReopenFile (ILTR_PTRANSL tr)
{
    int rc;
    PSTTIF_TYPE pstTIF;
    ILDFX_PHNDL phFile;

    /*-----
    * For a 16-bit translator running under 32-bit engine we
    * have to start from ground zero!!
    *-----*/
    if (ILTR_Flags & ILTR_FLAG_MIXED_WIN3216)
    {
        #ifdef ILX32_16
            rc = TIFInitStruct (tr);
            if (rc != SUCCESS)
                return rc;

            rc = TIFInit (&ILTIF_pstTIF, tr, -1); // special TIF init
            if (rc != SUCCESS)
                return rc;

            pstTIF = &ILTIF_pstTIF;
            TIFlogsz("ILTIFReopenFile/WinMix16");
            TIF_phase = TIF_PHASE_PREVIOUS;
            rc = TIFStartNextPhase (tr, ILTR_TifPhase);
            return rc;
        #endif
    }

    //---- for all other cases the TIF structs should exist

```

```

if (ILTR_pILTIF == NULL)
    return TIF_ERR_PILTIF_IS_NULL;

pstTIF = &ILTIF_pstTIF;
phFile = TIF_hFile;

/*-----
 * For unmixed Windows jobs we do nothing at all!
 * For Macintosh we simply reopen the file.
 *-----*/
if (ILTR_Flags & ILTR_FLAG_MACINTOSH)
{
    rc = ILLOG_reopen (TIFLOG);
    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_LOGGING);

    TIFlogsz("ILTIFReopenFile/Mac");
    rc = ILDFX_ReopenFile(TIF_szWorkFile, phFile, ILDFX_MODE_UPDATE);
    if (rc == SUCCESS)
    {
        TIF_bWorkFileIsOpen = TRUE;
        TIF_phase = TIF_PHASE_PREVIOUS;
        rc = TIFStartNextPhase (tr, ILTR_TifPhase);
    }
}

/*-----
 * For a MIXED Windows job, where a 32-bit engine is driving one or
 * more 16-bit translators, we need to do a lot more work. (The
 * code for 16-bit translators is above; here we handle 32-bit engine
 * or translator running in the MIXED 32/16 environment.)
 *-----*/
else if (ILTR_Flags & ILTR_FLAG_MIXED_WIN3216)
{
    rc = ILLOG_resume("tif.log", ILTR_hLog, &TIFLOG);
    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_LOGGING);

    TIFlogsz("ILTIFReopenFile/WinMix32");
    rc = ILDFX_OpenFile (TIF_szWorkFile, phFile, ILDFX_MODE_UPDATE);
    if (rc == SUCCESS)
    {
        TIF_bWorkFileIsOpen = TRUE;
        rc = ILDFX_GetRecordCount (TIF_hFile, &TIF_TotalRecordCount);
        if (rc == SUCCESS)
        {
            /*-----
             * Update misc params by reading TIF_RECORD_TWO of workfile
             *-----*/
            rc = TIFLoadKStruct (phFile, &((*pstTIF)->K));
            if (rc == SUCCESS)
            {
                TIF_phase = TIF_PHASE_PREVIOUS;
                rc = TIFStartNextPhase (tr, ILTR_TifPhase);
            }
        }
    }
}

else
{
    /*--- NO-OP for unmixed Windows environments (WIN16 and WIN32)
    TIFlogsz("ILTIFReopenFile/NO-OP");
    return SUCCESS;
}

ILTIFLOG = TIFLOG;          // for ILTIF-level logging
ILDFXLOG(TIF_hFile) = TIFLOG; // for ILDFX-level logging
return rc;

} //---- ILTIFReopenFile

/*-----
 * Name:      ILTIFBeginStatusBar (old name ILTIFBeginStatus)

```

```

* Purpose: Initialize and display the status bar
* Inputs:  tr
*          Global tr struct
*          szTheMsg
*          The string to be displayed
*          lNumRecs
*          Number of records
* Return:  SUCCESS      - If all went well.
* Author:  Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:
*-----*/
TIF_DLL_ENTRYPOINT ILTIFBeginStatusBar ( ILTR_PTRANSL tr,
                                          IL_PSTR szTheMsg,
                                          LONG lNumRecs )
{
    ILStatusInit ( tr,szTheMsg, lNumRecs );
    return SUCCESS;
} //---- ILTIFBeginStatusBar

//---- turn off aliasing so we can provide OLD entrypoint too
#undef ILTIFBeginStatus

extern "C" TIF_DLL_ENTRYPOINT ILTIFBeginStatus ( ILTR_PTRANSL tr,
                                                  IL_PSTR szTheMsg,
                                                  LONG lNumRecs )
{
    ILStatusInit ( tr,szTheMsg, lNumRecs );
    return SUCCESS;
} //---- ILTIFBeginStatus

/*-----*/
* Name:      ILTIFUpdateStatusBar (old name ILTIFUpdateStatus)
* Purpose: Increment the status bar
* Inputs:  tr
*          Global tr struct
* Return:  SUCCESS      - If all went well.
*          ILTR_ERR_CANCEL - If user cancelled
* Author:  Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:
*-----*/
TIF_DLL_ENTRYPOINT ILTIFUpdateStatusBar ( ILTR_PTRANSL tr )
{
    // Process all the messages in the queue and check for cancel
    int rc = ILProcessMessages(tr);
    if (rc == SUCCESS)
        ILStatusUpdate(tr); // Update the status bar 1 increment

    return rc;
} //---- ILTIFUpdateStatusBar

//---- turn off aliasing so we can provide OLD entrypoint too
#undef ILTIFUpdateStatus

extern "C" TIF_DLL_ENTRYPOINT ILTIFUpdateStatus ( ILTR_PTRANSL tr )
{
    // Process all the messages in the queue and check for cancel
    int rc = ILProcessMessages(tr);
    if (rc == SUCCESS)
        ILStatusUpdate(tr); // Update the status bar 1 increment

    return rc;
} //---- ILTIFUpdateStatus

/*-----*/
* Name:      ILTIFEndStatusBar (old name ILTIFEndStatus)
* Purpose: Take down and clean up the status bar
* Author:  Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*-----*/
TIF_DLL_ENTRYPOINT ILTIFEndStatusBar ( ILTR_PTRANSL tr )

```

```

(
    ILStatusDone ( tr );
    return SUCCESS;
} //---- ILTIFEndStatusBar

//---- turn off aliasing so we can provide OLD entrypoint too
#undef ILTIFEndStatus

extern "C" TIF_DLL_ENTRYPOINT ILTIFEndStatus ( ILTR_PTRANSL tr )
(
    ILStatusDone ( tr );
    return SUCCESS;
} //---- ILTIFEndStatus

/*-----
* Name:      ILTIFUnloadToILIF
* Purpose:   Unload all fields in one record from ILTIF to ILIF
* Input:     Pointers to translator data
* Return:    SUCCESS or error code
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
TIF_DLL_ENTRYPOINT ILTIFUnloadToILIF ( ILTR_PTRANSL tr )
(
    int rc = ILTIFReadNextRecord(tr);
    if (rc == TIF_EOF)
        return ILTR_EOF;
    else if (rc != SUCCESS)
        return rc;

    INT32 FieldCount;
    ILTIFHowManyField ( tr, &FieldCount );
    for ( INT32 fieldnum = 0; fieldnum < FieldCount; fieldnum ++ )
    {
        char szFieldName[ILTR_MAX_FLDNAME];
        ILTIFGetFieldName ( tr, fieldnum, szFieldName );

        //---- Don't process any TIF-internal fields (e.g. "\0x2InstanceArray")
        if (szFieldName[0] < 32)
            continue;

        IL_PSTR pField;
        LONG lField;
        rc = ILTIFGetField ( tr, szFieldName, TIF_CURRENT,
                           &lField, (IL_PANY *) &pField );
        switch (rc)
        {
            case SUCCESS:
            case ILTR_ERR_NODATA:
            case ILTR_ERR_NOTMAPPED:
            case ILTR_ERR_NOFLD:
            case ILTR_ERR_TRUNC:
                break;                // tolerate these errors

            case ILTR_ERR_NOMEM:
            case ILTR_ERR_FILE:
            default:
                return rc;            // complain about other errors
        }

        // Place the field in the intermediate file
        ILFldPut (tr, szFieldName, pField, IL_STRLEN(pField));
    }

    return SUCCESS;
} //---- ILTIFUnloadToILIF

/*-----
* Name:      ILTIFHowManyRecords
* Purpose:   Return the number of records in the TIF mechanism

```



```

* Inputs:  tr
*          Global tr struct
*          lNumOfRecs
*          Pointer to number of records in the mechanism
* Return:  SUCCESS          - If all went well.
*          lNumRecs filled in
* Author:  Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:
*-----*/
TIF_DLL_ENTRYPOINT ILTIHowManyRecords ( ILTR_PTRANSL tr,
                                         LONG *lNumOfRecs )
{
    if (ILTR_pILTIF == NULL)
        return ILERROR (0, TIF_ERR_PILTIF_IS_NULL);

    //--- set limit to stop forward scanning of TIF index when unloading
    ILTI_pstTIF->GoalPost = ILTI_pstTIF->GoalPostForNextPass;

    *lNumOfRecs = ILTI_pstTIF->PertinentRecordCount;

    return SUCCESS;
} //---- ILTIHowManyRecords

/*-----*/
* TIFPutSourceRecord -- used in ILX_V4 Phase 20
*
* The source translator, operating under ILX_V4, is loading
* data that is subject to Field Mapping. Values for Unmapped
* Source Fields are already in TIF_CurrentRecord, but
* values for Mapped Source Fields are in TIF_SourceRecord.
* We now do Field Mapping to construct corresponding
* Target Field Values, which are put into TIF_CurrentRecord,
* then finally we store TIF_CurrentRecord.
*-----*/
int TIFPutSourceRecord (ILTR_PTRANSL tr)
{
    int rc;
    PSTTIF_TYPE pstTIF = &ILTR_pILTIF->pstTIF;
    long lFieldCount;
    long fldnum;
    long FieldLength;
    IL_PANY FieldValue;

    if (ILTR_phase != ILTR_PHASE20)
        return ILERROR (ILTR_phase, TIF_ERR_ABNORMAL);

    else // exporting from SOURCE ... need to do field mapping
    {
        rc = ILTIHowManyField (tr, &lFieldCount);
        if (rc != SUCCESS)
            return ILERROR(rc, TIF_ERR_ABNORMAL);

        for (fldnum=0; fldnum < lFieldCount; fldnum++)
        {
            if (TIF_FieldIsMapped(pstTIF, fldnum))
            {
                IL_PSTR fldname = TIF_FieldName(pstTIF, fldnum);
                rc = GetMappedField ( tr, fldname, TIF_SOURCE_CACHE,
                                     FALSE, //--- don't do character mapping
                                     &FieldLength, &FieldValue );
                if (rc != SUCCESS)
                    return ILERROR(rc, TIF_ERR_ABNORMAL);

                //---- Put field value in TIF Current Record Value struct
                rc = TIFRecordAddFieldValue ( pstTIF,
                                              TIF_pFieldList,
                                              fldname, TIF_NOTSET,
                                              (IL_PSTR) FieldValue,
                                              FieldLength,
                                              FALSE, // don't do character mapping
                                              &TIF_CurrentRecord );

                if (ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 98))

```

```

        TIFLogPutField(tr, fldname, (UINT8 *) FieldValue, FieldLength, rc);

        if (rc != SUCCESS)
            return ILERROR(rc, TIF_ERR_ABNORMAL);
    }
} // end for
} // end else

/*-----
 * Write Source Record to TIF
 *
 * If Target Translator wants to "sanitize" Source Records, we simply store
 * an un-analyzed record here. Analysis is deferred until this Sanitizing
 * step.
 *
 * But if Target Translator is NOT going to sanitize Source Records, we
 * need to do the record analysis now.
 *-----*/
TIF_PUT_RECORD_OPTION opt;

if (ILTR_Flags & ILTR_FLAGS_SKIP_SANITIZING_STEP)
    opt = TIFPRO_ANALYZE_AND_STORE_NEW_RECORD;
else
    opt = TIFPRO_STORE_NEW_UNANALYZED_RECORD;

rc = TIFPutRecord (tr, opt);
if (rc != SUCCESS)
{
    //----- log the error
    TIFlogszul ("PutSourceRecord rc=%ld", (UINT32) rc);

    //----- look for no memory condition
    if (rc == ILDFX_ERR_NOMEM)
        rc = ILERROR(rc, ILTR_ERR_NOMEM);
    else
        rc = ILERROR(rc, TIF_ERR_ABNORMAL);
}

return rc;
} //---- TIFPutSourceRecord

/*-----
 * Name:      ILTIFDontSyncByID
 * Purpose: Tells the Synchronization Engine not to use Unique IDs in doing
 *           synchronization.
 *
 * NOTE:      calling this function is equivalent to calling
 *            ILTIFFeatureSet (tr, 0, TIF_DISABLE_SYNC_BY_ID);
 *
 * Comments: This call turns off usage of unique IDs for sync, but TIF still
 *           does ordinary storage and retrieval of unique IDs, if unique ID
 *           fields are defined.
 *
 *           This function is only needed when a translator that has a field
 *           in its field list called _UniqueID wants TIF to do KeyField-based
 *           synchronization rather than uniqueID-based synchronization.
 *
 *           For a synchronization job involving translators ILXAAA and ILXBBB,
 *           the field lists of one or the other or both or neither of ILXAAA
 *           and ILXBBB may have _UniqueID fields. If both have _UniqueID
 *           fields, then if ILXAAA calls ILTIFDontSyncByID the sync engine
 *           will do KeyField-based correlation on the ILXAAA side of things,
 *           but correlation on the ILXBBB side will still be done by Unique ID.
 *
 *           TIF requires that this function be called either during the
 *           load-from-target phase or during the load-from-source phase.
 *
 *           However TIF does not apply any other restrictions on when this
 *           function may be called. Unless wacky counter-arguments suggest
 *           otherwise, a translator should call ILTIFDontSyncByID before
 *           putting any records into TIF. Suggestion: call it in your
 *           BEGIN routine when ILTR_direction == ILTR_EXPORT.
 *-----

```

```

* Author: David Boothby, Copyright (c) IntelliLink Corporation.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFDontSyncByID (ILTR_PTRANSI tr)
{
    return ILTIFFeatureSet (tr, 0, TIF_DISABLE_SYNC_BY_ID);
} //---- ILTIFDontSyncByID

/*-----
* Name: ILTIFFeatureSet
* Purpose: Tells TIF to turn specified feature bits ON or OFF.
*
* Comments: This function has a general-purpose interface, but initially
*           it's only use is to disable FastSync and/or sync-by-uniqueID
*           for the current phase.
*
* NOTE: Sync-by-uniqueID is a pre-requisite for FastSync, so if sync-by-ID
*       is disabled then FastSync is also disabled.
*
* Author: David Boothby, Copyright (c) IntelliLink Corporation.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFeatureSet ( ILTR_PTRANSI tr,
                                     UINT32 ulTurnOnBits,
                                     UINT32 ulTurnOffBits )
{
    int rc = SUCCESS;
    UINT32 ulOff;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    ulOff = ulTurnOffBits & (TIF_DISABLE_FAST_SYNC | TIF_DISABLE_SYNC_BY_ID);

    if (ulTurnOnBits != 0 || ulTurnOffBits != ulOff)
        //---- assume that a newer caller is calling an older TIF...
        rc = TIF_ERR_NOT_YET_IMPLEMENTED;

    else if (ulOff == 0)
        rc = TIF_ERR_BAD_PARAM;

    else
    {
        PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

        switch (TIF_phase)
        {
            case TIF_PHASE_LOADING_TARGET_RECORDS:

                if (ulOff & TIF_DISABLE_SYNC_BY_ID)
                    TIF_bSyncUsingTargetIDs = FALSE;
                TIF_bFastSyncTargetLoad = FALSE;
                TIF_bFastSyncLoad = FALSE;
                break;

            case TIF_PHASE_LOADING_SOURCE_RECORDS:

                if (ulOff & TIF_DISABLE_SYNC_BY_ID)
                    TIF_bSyncUsingSourceIDs = FALSE;
                TIF_bFastSyncSourceLoad = FALSE;
                TIF_bFastSyncLoad = FALSE;
                break;

            default:
                rc = TIF_ERR_BAD_STATE;
        }

        if (rc == SUCCESS)
        {
            /*-----
            * Update Record TWO of Work File. This ensures that the modified
            * flags will persist across close/reopen actions on the workfile.
            *-----*/
            rc = ILDFX_UpdateRecord ( TIF_hFile,
                                     TIF_RECORD_TWO,

```

```

        &((*pstTIF)->K),
        sizeof(TIF_KSTRUCT),
        NULL ); //-- no ExData for this record

    if (rc != ILDFX_OK)
        return rc;
}

ILTIFlogsul3 ( "ILTIFFeatureSet(0x%lx, 0x%lx) rc=%ld",
               ulTurnOnBits, ulTurnOffBits, (UINT32) rc);
return rc;
} //---- ILTIFFeatureSet

/*-----
 * ILTIFItemIsRecurring
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFItemIsRecurring (ILTR_PTRANSL tr)
{
    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;
    else
    {
        PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;
        INT32 CurrentRecNum;
        INT32 OriginalRecNum;
        int rc = TIFSetRecordNumbers ( pstTIF, TIF_CurrentRecordNumber,
                                       &CurrentRecNum, &OriginalRecNum );

        if (rc != SUCCESS)
            return rc;

        if (CurrentRecNum == TIF_NOTSET)
            CurrentRecNum = OriginalRecNum;

        if (CurrentRecNum == TIF_NOTSET)
            return ILERROR(-1, TIF_ERR_BAD_STATE);

        return TIFX_ITEM_IS_RECURRING (TIF_hFile, CurrentRecNum);
    }
} //---- ILTIFItemIsRecurring

/*-----
 * Name:      PutAdjustedDates
 *
 * called from ILTIFFanItem
 *-----*/
static int PutAdjustedDates ( PSTTIF_TYPE pstTIF,
                              INT32 lDate, ILUT_PBUFFER pRecBuf )
{
    char szDate[10];
    int rc;

    //---- Convert encoded date to YYYYMMDD
    IL_CodeDateToAlpha (lDate, szDate);

    //---- Adjust Start Date if Start Date is ABSOLUTE
    if ( TIF_StartDateFieldNum != TIF_NOTSET
        && TIF_FieldType(pstTIF, TIF_StartDateFieldNum) == ILX_TYPE_DATE )
    {
        rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList,
                                      "", TIF_StartDateFieldNum,
                                      szDate, 0,
                                      FALSE, // do no character mapping
                                      pRecBuf );

        if (rc != SUCCESS)
            return ILERROR(rc, rc);
    }

    //---- Adjust End Date if End Date is ABSOLUTE
    if ( TIF_EndDateFieldNum != TIF_NOTSET
        && TIF_FieldType(pstTIF, TIF_EndDateFieldNum) == ILX_TYPE_DATE )
    {

```

```

        rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList,
                                      "", TIF_EndDateFieldNum,
                                      szDate, 0,
                                      FALSE, // do no character mapping
                                      pRecBuf );

        if (rc != SUCCESS)
            return ILERROR(rc, rc);
    }

    //---- Adjust Alarm Date if Alarm Date is ABSOLUTE
    if ( TIF_AlarmDateFieldNum != TIF_NOTSET
        && TIF_FieldType(pstTIF, TIF_AlarmDateFieldNum) == ILX_TYPE_DATE )
    {
        rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList,
                                      "", TIF_AlarmDateFieldNum,
                                      szDate, 0,
                                      FALSE, // do no character mapping
                                      pRecBuf );

        if (rc != SUCCESS)
            return ILERROR(rc, rc);
    }

    return SUCCESS;
} //---- PutAdjustedDates

/*-----
 * Name:      MarkAllRecurringCigMembers
 * Purpose:   set some bits for all CIG members that are recurring items.
 * Called by: ILTIFFanItem to mark all masters FANNED for SOURCE or for TARGET
 *-----*/
static int MarkAllRecurringCigMembers ( ILDFX_PHNDL phFile, INT32 Item,
                                       INT32 FlagsToSet )
{
    if (TIFX_ITEM_IS_RECURRING(phFile, Item))
        TIFX_FLAGS(phFile, Item) |= FlagsToSet;

    INT32 Next = Item;
    int i;
    for (i=1; i <= TIF_MAX_CIG_SIZE; i++) // infinite loop protection
    {
        Next = TIFX_NEXT_IN_CIG(phFile, Next);
        if (Next == Item)
            break;

        if (TIFX_ITEM_IS_RECURRING(phFile, Next))
            TIFX_FLAGS(phFile, Next) |= FlagsToSet;
    }

    if (i > TIF_MAX_CIG_SIZE)
        return ILERROR_L(Item, TIF_ERR_BROKEN_CIG); // too big or not circular

    return SUCCESS;
} //---- MarkAllRecurringCigMembers

/*-----
 * LogFig
 *-----*/
static void LogFig (ILDFX_PHNDL phFile, INT32 Start, IL_PSTR szRole)
{
    char szBuf[120];
    char szTmp[40];
    int count=0;
    INT32 Origin;
    IL_PSTR szOrigin;
    INT32 Next;

    if (Start == TIF_NOTSET)
    {
        TIFXlogszsz ("%s not present", szRole);
        return;
    }
}

```

```

Origin = TIFX_ORIGIN (phFile, Start);
switch (Origin)
{
    case TIF_FROM_PREVIOUS : szOrigin = "Previous"; break;
    case TIF_FROM_TARGET   : szOrigin = "Target";   break;
    case TIF_FROM_SOURCE   : szOrigin = "Source";   break;
}

Next = TIFX_NEXT_IN_FIG(phFile, Start);
if (Next == Start)
{
    TIFXlogsz3ul ( "%s (%s) item (%ld) is a singleton FIG",
                  szOrigin, szRole, Start );
    return;
}

TIFXlogsz3ul ( "%s (%s) item (%ld) -- FIG analysis:",
              szOrigin, szRole, Start );

IL_SPRINTF (szBuf, "FIG: %04ld", Start);

while (Next != Start)
{
    IL_SPRINTF (szTmp, ", %04ld", Next);
    IL_STRCAT (szBuf, szTmp);
    count++;
    if (count % 10 == 0)
    {
        TIFXlogsz (szBuf);
        IL_STRCPY (szBuf, " ");
    }

    Next = TIFX_NEXT_IN_FIG(phFile, Next);
}

TIFXlogsz (szBuf);
} //---- LogFig

/*-----
* Name:      ILTIFFanItem
*
*           Item to fan is record number TIF_lCurrentRecNum
*
*           This function is only called recurring items whose current
*           unload outcome is ADD or UPDATE.
*
* Author:    David Boothby, Copyright (c) IntelliLink Corporation.
*-----*/
TIF_DLL_ENTRYPOINT ILTIFFanItem (ILTR_PTRANSL tr, int maxFanCount)
{
    int rc = FanItem (tr, maxFanCount);
    if (rc == SUCCESS && ILLOG_VERBOSE_ENOUGH(ILTIFLOG, 99))
    {
        PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;
        ILDFX_PHNDL phFile = TIF_hFile;
        INT32 Master = TIF_lCurrentRecNum;
        INT32 cig[3];
        INT32 spt[3];
        INT32 Updater;           // Item that instigated the UPDATE
        INT32 Updatee;          // Item that is to be UPDATED
        INT32 Middleman;        // Previous (History) Item
        INT32 Next;
        char szText[100];

        //---- Identify the Source, Previous, and Target members of the CIG
        int rc = TIFBuildSPT (phFile, Master, cig, spt);
        if (rc != SUCCESS)
            return ILERROR_L (Master, rc);

        //---- Identify the Updater and Updatee items
        if (TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET)
    {

```

```

        Updater = spt[TIF_SPT_S];    // source side is instigator
        Updatee = spt[TIF_SPT_T];
    }
    else
    {
        Updater = spt[TIF_SPT_T];    // target side is instigator
        Updatee = spt[TIF_SPT_S];
    }

    //---- Identify Middleman (who may or may not exist!!!)
    Middleman = spt[TIF_SPT_P];

    //---- Guard against the impossible
    if (Updater == TIF_NOTSET)
        return ILERROR_L (Master, TIF_ERR_ABNORMAL);

    //---- Look for Fanned Instances of the Updater
    Next = TIFX_NEXT_IN_FIG(phFile, Updater);

    IL_SPRINTF ( szText, "Fanning completed for: %ld --(%ld)--> %ld, %s",
                  Updater, Middleman, Updatee,
                  TIFCigName (TIFX_CIG_TYPE(phFile, Updater)) );
    TIFlogsz (szText);

    LogFig (phFile, Updater, "Other");
    LogFig (phFile, Middleman, "Middle");
    LogFig (phFile, Updatee, "Unloader");

    }

    return rc;
} //---- ILTIFFanItem

/*-----
* Name:      FanItem
*
*           Item to fan is record number TIF_lCurrentRecNum
*
*           This function is only called recurring items whose current
*           unload outcome is ADD or UPDATE.
*
* Author:    David Boothby, Copyright (c) IntelliLink Corporation.
*-----*/
static int FanItem (ILTR_PTRANSL tr, int maxFanCount)
{
    int rc;
    int i;
    int fannedCount;
    int FieldCount;
    int fldnum;
    ILTR_PREPEAT pRepeat = NULL;
    int FullMasterCount;
    int UnexcludedMasterCount;
    TIFSYNC_INSTANCE_TYPE *MasterArray = NULL;
    ILTR_PFANOUT_MAXIMA pMaxima;
    PSTTIF_TYPE pstTIF;
    ILDFX_PHNDL phFile;
    INT32 Master;
    INT32 Instance;
    INT32 Next;
    INT32 FannedForWhom;
    INT32 MasterFlags;
    INT32 InstanceFlags;
    INT32 SavedOutcome;
    INT32 Outcome;
    INT32 UpdaterIDFldNum;
    INT32 UpdateeIDFldNum;
    INT32 UpdateeIDHashSlot;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    pstTIF = &ILTIF_pstTIF;

```

```

phFile = TIF_hFile;
Master = TIF_lCurrentRecNum;
MasterFlags = TIFX_FLAGS (phFile, Master);

//---- get the internal outcome flags for the CIG containing the record
INT32 CigOutcome = MasterFlags & TIF_OUTCOME_SYNC_MASK;

//---- get the unload outcome for the current record
rc = ILTIFGetOutcome (tr, &Outcome);
if (rc != SUCCESS)
    return ILERROR (rc, rc);

//---- if outcome isn't ADD or UPDATE we don't do fanning here.
if ((Outcome & (ILTIF_OUTCOME_ADD | ILTIF_OUTCOME_UPDATE)) == 0)
    return ILERROR_L (Outcome, TIF_ERR_CANT_FAN_THIS_OUTCOME);

/*-----
 * Aug 6, 1996: Here we rely on a brute-force behavior of TIF, which is
 * that when we need to update a previously FANNED item, we do that by
 * deleting all the previously fanned instances and then re-fanning the
 * master to generate new instances (in effect we do a REPLACE operation)
 * This behavior is implemented by the TIFTableGetOutcomeForSync function
 * in tiftable.cpp.
 * Based on that knowledge of TIF's behavior, we know that whenever we
 * see an UPDATE outcome here, for a recurring item, that recurring
 * item has NOT previously been fanned to the current unloading side.
 *-----*/

//---- Determine what flag bits to set for the Fanned Instance Items.
if (TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET)
{
    FannedForWhom = TIF_IS_FANNED_FOR_TARGET;
    UpdaterIDFldNum = TIF_SourceIDFieldNum;
    UpdateeIDFldNum = TIF_TargetIDFieldNum;
    UpdateeIDHashSlot = TIF_TARGETID_HASH_SLOT;
    pMaxima = &TIF_TargetFanoutMaxima;
}
else if (TIF_phase == TIF_PHASE_UNLOADING_TO_SOURCE)
{
    FannedForWhom = TIF_IS_FANNED_FOR_SOURCE;
    UpdaterIDFldNum = TIF_TargetIDFieldNum;
    UpdateeIDFldNum = TIF_SourceIDFieldNum;
    UpdateeIDHashSlot = TIF_SOURCEID_HASH_SLOT;
    pMaxima = &TIF_SourceFanoutMaxima;
}
else
    return ILERROR(TIF_phase, TIF_ERR_BAD_STATE);

//--- set origin and CIG type of instances to be same as master
InstanceFlags = MasterFlags & (TIF_ORIGIN_MASK | TIF_CIG_TYPE_MASK);
InstanceFlags |= FannedForWhom;

/*-----
 * Read current record into memory (probably redundantly).
 * Then fan out recurrence pattern to build master array. Also get a
 * copy of the BASIC recurrence pattern back in 'pRepeat'. This function
 * is responsible for freeing what 'pRepeat' points to, but it is NOT
 * responsible for worrying about there being an exclusion list to free.
 *-----*/
rc = TIFSyncFanOutRecurrencePattern ( pstTIF, Master, pMaxima,
                                     &MasterArray,
                                     &pRepeat,
                                     FALSE, // don't merge new+old fanouts
                                     &FullMasterCount,
                                     &UnexcludedMasterCount );

if (pRepeat != NULL)
    delete pRepeat;

if (rc != SUCCESS)
    LOG_ERR_AND_EXIT (rc, TIF_ERR_CANT_FAN)

if (FullMasterCount == 0)
    //--- master generates no instances ... unusual ...
    EXIT_WITH_ERROR (SUCCESS);

```



```

//---- for UPDATE outcomes we need to do some very special surgery
if (Outcome & ILTIF_OUTCOME_UPDATE)
{
    rc = FanItemUpdate (pstTIF, &Master);
    if (rc != SUCCESS || Master == TIF_NOTSET)
    {
        if (MasterArray != NULL) delete MasterArray;
        return rc;
    }
}

/*-----
 * Discard any old Fanned Instances attached to the Master
 *-----*/

Next = TIFX_NEXT_IN_FIG(phFile, Master);
for (i=1; i <= TIF_MAX_FIG_SIZE; i++) // infinite loop protection
{
    if (Next == Master)
        break;
    else
    {
        Instance = Next;
        TIFloglint (90, "Marking old FIG member #%ld as garbage", Instance);
        TIFX_FLAGS(phFile, Instance) |= TIF_IS_GARBAGE;
        Next = TIFX_NEXT_IN_FIG(phFile, Instance);
        TIFX_NEXT_IN_FIG(phFile, Instance) = Instance; // singleton
    }
}

//---- croak if FIG is too big or isn't circular
if (i > TIF_MAX_FIG_SIZE)
    return ILERROR_L(Master, TIF_ERR_BROKEN_FIG);

//---- Now Master belongs to a singleton FIG
TIFX_NEXT_IN_FIG(phFile, Master) = Master;

/*-----
 * If the "updatee" system uses Unique IDs, put special IDHASH value into
 * the "updater" Master item to indicate that an ID-bearing FIG is attached
 * to it.
 *-----*/
if (UpdateeIDFldNum != TIF_NOTSET)
    TIFX(phFile, Master, UpdateeIDHashSlot) = TIFHASH_SPECIAL;

/*-----
 * Use 'TIF_FirstRecord' buffer for building Fanned Instances.
 *-----*/
rc = TIFInitRecord (pstTIF, TIF_pFieldList, &TIF_FirstRecord);
if (rc != SUCCESS)
    LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

/*-----
 * Copy all non-repeat info from master into 'FirstRecord' buffer.
 *-----*/
FieldCount = TIF_FieldCount(pstTIF);
for (fldnum = 0; fldnum < FieldCount; fldnum++)
{
    /*-----
     * Don't put REPEAT info into fanned instances, and don't copy
     * "other ID" into fanned instances.
     *-----*/
    if ( (fldnum == TIF_RepBasicFieldNum)
        || (fldnum == TIF_RepExclFieldNum)
        || (fldnum == UpdaterIDFldNum) )
        continue;

    if (TIF_FieldOffset(TIF_pCurrentRecord, fldnum) != TIF_NOTSET)
    {
        INT32 len = TIF_FieldLength(TIF_pCurrentRecord, fldnum);
        IL_HPSTR p = TIF_FieldData(TIF_pCurrentRecord, fldnum);
        if (len > 0)
        {
            rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList,
                                         "", fldnum,

```

```

        p, len,
        FALSE, // do no character mapping
        &TIF_FirstRecord );
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);
}

/*-----
 * Generate and store fanned instances
 *-----*/
INT32 recnum; recnum = TIF_NOTSET;
fannedCount = 0;
for (i=0; i < FullMasterCount; i++)
{
    TIF_RECORD_VALUE_PTR pRec;
    INT32 exdata[TIF_EXDATA_PER_RECORD];
    INT32 InstanceDate;
    char szDateDisplay[50];

    if (MasterArray[i].bExcludedFromMaster)
        continue;

    // Increment the number of records in the file
    recnum = TIF_TotalRecordCount++;

    InstanceDate = MasterArray[i].lDate;
    TIFlog11 ( 60, "Creating Fanned Instance for %s, recnum=%ld",
              IL_CodeDateToStdDisplay (InstanceDate, szDateDisplay),
              recnum );

    //---- put adjusted dates into *TIF_pFirstRecord
    rc = PutAdjustedDates (pstTIF, InstanceDate, &TIF_FirstRecord);
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

    IL_MEMSET( (IL_PANY) exdata, 0, sizeof(exdata));

    exdata[TIF_FLAGS_SLOT] = InstanceFlags;
    exdata[TIF_NEXT_IN_CIG_SLOT] = recnum; // singleton CIG
    exdata[TIF_NEXT_IN_SKG_SLOT] = recnum; // singleton SKG
    exdata[TIF_NEXT_IN_FIG_SLOT] = recnum; // singleton FIG

    pRec = TIF_pFirstRecord;

    //---- compute hash values and start/end DTTM values
    rc = TIFComputeSearchKeyValues(pstTIF, pRec, exdata);
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

    rc = ILDFX_AddRecord (phFile, recnum, pRec, TIFREC_SIZE(pRec), exdata);
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

    TIFX_NEXT_IN_FIG(phFile, recnum) = TIFX_NEXT_IN_FIG(phFile, Master);
    TIFX_NEXT_IN_FIG(phFile, Master) = recnum;

    fannedCount++;
}

/*-----
 * By virtue of the fanning we've just done we've added a bunch of
 * pertinent records.
 *
 * NOTE: this new count doesn't take effect until the next time the
 * user calls ILTIFHowManyRecords.
 *-----*/
TIF_PertinentRecordCount += fannedCount;

/*-----
 * For UPDATE, the recurring master record is still pertinent (its
 * outcome is now DELETE). But for ADD the recurring master isn't
 * pertinent at all, so here we must reduce the count by 1.
 *-----*/

```

```

    if (Outcome & ILTIF_OUTCOME_ADD)
        TIF_PertinentRecordCount -= 1;

    //---- push goalpost out for next unload scan but not for this scan
    if (recnum != TIF_NOTSET)
        TIF_GoalPostForNextPass = recnum;

    /*-----
    * Mark the Master as Fanned for Source or Fanned for Target.
    *-----*/
    rc = MarkAllRecurringCigMembers (phFile, Master, FannedForWhom);
    if (rc != 0)
        LOG_ERR_AND_EXIT (rc, TIF_ERR_ABNORMAL);

    /*-----
    * Put FAN entry in xlate.log
    *-----*/
    SavedOutcome = TIF_CurrentRecordOutcome;
    TIF_CurrentRecordOutcome = ILTIF_OUTCOME_FANNED;
    rc = LogRecord (tr);

    TIF_CurrentRecordOutcome = SavedOutcome;

Exit:

    if (MasterArray != NULL) delete MasterArray;

    ILTIFlogszulul ( "ILTIFFanItem count=%ld, rc=%ld",
                    (UINT32) fannedCount, (UINT32) rc );

    return rc;
} //---- FanItem

/*-----
* Name:      FanItemUpdate
*-----*/
static int FanItemUpdate (PSTTIF_TYPE pstTIF, INT32 *pMaster)
{
    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 Master = *pMaster;
    INT32 cig[3];
    INT32 spt[3];
    INT32 Updater;          // Item that instigated the UPDATE
    INT32 Updatee;          // Item that is to be UPDATED
    INT32 Middleman;        // Previous (History) Item
    INT32 Next;
    BOOLEAN bChopItUp;

    //---- Identify the Source, Previous, and Target members of the CIG
    int rc = TIFBuildSPT (phFile, Master, cig, spt);
    if (rc != SUCCESS)
        return ILERROR_L (Master, rc);

    //---- Identify the Updater and Updatee items
    if (TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET)
    {
        Updater = spt[TIF_SPT_S];    // source side is instigator
        Updatee = spt[TIF_SPT_T];
    }
    else
    {
        Updater = spt[TIF_SPT_T];    // target side is instigator
        Updatee = spt[TIF_SPT_S];
    }

    //---- Identify Middleman (who may or may not exist!!!)
    Middleman = spt[TIF_SPT_P];

    //---- Guard against the impossible
    if (Updater == TIF_NOTSET)
        return ILERROR_L (Master, TIF_ERR_ABNORMAL);

    //---- Look for Fanned Instances of the Updater
    Next = TIFX_NEXT_IN_FIG(phFile, Updater);

```

```

if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 99))
{
    char szText[100];
    IL_SPRINTF ( szText, "FanUpdate: %ld/%ld --(%ld)--> %ld, %s",
        Updater, Next, Middleman, Updatee,
        TIFCigName (TIFX_CIG_TYPE(phFile, Updater)) );
    TIFlogsz (szText);
}

/*-----
 * Decide whether or not to chop the CIG into bits. Usually we do, but if
 * we're doing standard fanning during the UNLOAD-to-TARGET phase for an
 * UPDATE-BOTH CIG, we keep the CIG together and give it a new CIG type
 *-----*/
bChopItUp = FALSE;

//---- If the Updater is fanned, we chop the CIG
if (Next != Updater)
    bChopItUp = TRUE;

//---- If CIG type isn't 132, we chop the CIG
else if (TIFX_CIG_TYPE(phFile, Updater) != TIF_CIG_TYPE_132)
    bChopItUp = TRUE;

//---- If we're unloading to Source, we chop the CIG.
else if (TIF_phase == TIF_PHASE_UNLOADING_TO_SOURCE)
    bChopItUp = TRUE;

//---- Do the CIG-chopping approach if required
if (bChopItUp)
{
    rc = FanItemCigChop (pstTIF, Updater, Updatee, Middleman);
    if (rc != SUCCESS)
        return rc;

    //---- If the Updater is fanned, we're all done now
    if (Next != Updater)
    {
        *pMaster = TIF_NOTSET;
        return SUCCESS;
    }
}

//---- If not doing the "CIG chop", set CIG type to "132 Fanned Target"
else
{
    TIFloglint (80, "Setting CIG type for %ld, et. al. to 13F", Updater);
    SetCigType (phFile, Updater, TIF_CIG_TYPE_13F);
    SetCigType (phFile, Updatee, TIF_CIG_TYPE_13F);
    SetCigType (phFile, Middleman, TIF_CIG_TYPE_13F);
}

/*-----
 * Now that we have chopped up the CIG, the rest of this UPDATE operation
 * is just like an ADD operation.
 *-----*/
TIFlog2ints ( 80, "Replacing Master %ld; will fan %ld to create instances",
    Updatee, Updater );
*pMaster = Updater;
return SUCCESS;
} //---- FanItemUpdate

/*-----
 * Name:      FanItemCigChop
 *-----*/
static int FanItemCigChop ( PSTTIF_TYPE pstTIF, INT32 Updater,
    INT32 Updatee, INT32 Middleman )
{
    ILDEX_PHNDL phFile = TIF_hFile;
    INT32_newCIGType;
    INT32_DeleteUpdateeBit;
    int rc;

```

```

TIFlog3ints ( 80, "FanItemCigChop %ld %ld %ld",
              Updater, Updatee, Middleman );

//---- bitmask used to clear away old bits from items...
#define UCLRMASK ( TIF_OUTCOME_SYNC_MASK \
                  - | TIF_CIG_TYPE_MASK \
                    | TIF_IS_FANNED_FOR_TARGET \
                    | TIF_IS_FANNED_FOR_SOURCE \
                    | TIF_KFM_ANY12 \
                    | TIF_IS_GOBBLED_UP_INSTANCE )

if (TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET)
{
    newCIGType = TIF_CIG_TYPE_100; //---- added on Source Side
    DeleteUpdateeBit = TIF2_DELETE_TARGET;
}
else
{
    newCIGType = TIF_CIG_TYPE_001; //---- added on Target Side
    DeleteUpdateeBit = TIF2_DELETE_SOURCE;
}

//---- Break the CIG into 3 separate singletons
TIFX_NEXT_IN_CIG(phFile, Updater) = Updater;
TIFX_NEXT_IN_CIG(phFile, Updatee) = Updatee;
if (Middleman != TIF_NOTSET)
    TIFX_NEXT_IN_CIG(phFile, Middleman) = Middleman;

//---- Mark the Middleman and any instances attached to it as garbage
if (Middleman != TIF_NOTSET)
{
    rc = TIFMarkAllFigMembers (phFile, Middleman, 0, TIF_IS_GARBAGE, 0);
    if (rc != SUCCESS) return ILERROR_L (Middleman, rc);
}

//---- Make the Updater Master and any Updater Fanned Instances,
//---- into ADDs, and clear other bits that no longer apply.
rc = TIFMarkAllFigMembers (phFile, Updater, UCLRMASK, newCIGType, 0);
if (rc != SUCCESS) return ILERROR_L (Updater, rc);

//---- Make the Updatee Master into a "DELETE-ME" Singleton
TIFX_FLAGS2(phFile, Updatee) |= DeleteUpdateeBit;

/*-----
 * If the Updater is fanned then we don't actually do a traditional
 * fanning operation at all. Instead we get rid of all the recurring
 * masters, by chopping off the top-level CIG, then we turn all the
 * Updater's fanned instances into ADDs. If the HistoryMaster has
 * any fanned instances hanging off it then they become garbage.
 * BUG: when we do this "chopping off" thing while fanning to TARGET we
 * lose track of any need to update the SOURCE. So an UPDATE-BOTH
 * outcome that passes through this code will NOT be fully executed.
 *-----*/
INT32 Next; Next = TIFX_NEXT_IN_FIG(phFile, Updater);
if (Next != Updater)
{
    //---- find out how many fanned instances there are in the FIG
    int fannedCount = 0;
    int i;
    for (i=1; i <= TIF_MAX_FIG_SIZE; i++) // infinite loop protection
    {
        fannedCount++;
        Next = TIFX_NEXT_IN_FIG(phFile, Next);
        if (Next == Updater)
            break;
    }

    //---- croak if FIG is too big or isn't circular
    if (i > TIF_MAX_FIG_SIZE)
        return ILERROR_L(Updater, TIF_ERR_BROKEN_FIG);

    TIFlog3ints ( 80,
                  "Removing masters %ld,%ld; %d instances will ADD across",
                  Updater, Updatee, fannedCount );
}

```

```

        //----- Turn the Updater (synthetic) Master into a garbage singleton.
        //----- The Updater Fanned Instances will be added
        TIFX_FLAGS(phFile, Updater) |= TIF_IS_GARBAGE;

        //----- Adjust the pertinent record count by however many fanned
        //----- instances we'll be adding. Note that the MASTER that we'll
        //----- be deleting is still included in the pertinent record count.
        TIF_PertinentRecordCount += fannedCount;
    }

    return SUCCESS;
} //----- FanItemCigChop

/*-----
 * Name:      SetCigType
 *-----*/
static void SetCigType ( ILDFX_PHNDL phFile,
                        INT32 Item,
                        INT32 CigType )
{
    TIFX_FLAGS (phFile, Item) &= ~TIF_CIG_TYPE_MASK;
    TIFX_FLAGS (phFile, Item) |= CigType;
}

/*-----
 * ILTIFRemoveRecord
 *
 * To implement a 'chooser' function, for SmartMerge, you must call
 * ILTIFStartNextPhase (tr, TIF_PHASE_CHOOSING_RECORDS), just before calling
 * ILTIFEndLoad; then you can read the incoming (source) records, just as if
 * you were in the UNLOAD-to-TARGET phase, and then for any records that you
 * want to remove (so that they aren't visible to the conflict resolution
 * and unload-to-target phases) you simply call ILTIFRemoveRecord.
 *
 * If you want to remove the "current" record, pass lRecNum = -1.
 *
 * For example:
 *
 *         rc = ILTIFReadNextRecord (tr);
 *         rc = ILTIFRemoveRecord (tr, -1);
 *
 * If you want to remove an arbitrary record, pass actual recnum.
 *
 * For example:
 *
 *         rc = ILTIFReadNextRecord (tr);
 *         rc = ILTIFRecordNum (tr, &lRecNum);
 *         rc = ILTIFRemoveRecord (tr, lRecNum);
 *
 * When your 'chooser' function is finished, call ILTIFEndLoad, etc.
 *
 * Records that you have 'removed' will show up in TIF.LOG with
 * ODD Flags values (the GARBAGE bit is 0x00000001).
 *-----*/
TIF_DLL_ENTRYPOINT ILTIFRemoveRecord (ILTR_PTRANSL tr, INT32 lRecNum)
{
    int rc;
    int loglevel;
    PSTTIF_TYPE pstTIF;
    ILDFX_PHNDL phFile;

    if (ILTR_pILTIF == NULL)
        return TIF_ERR_PILTIF_IS_NULL;

    pstTIF = &ILTIF_pstTIF;
    phFile = TIF_hFile;

    //----- make sure we're in the right phase for doing this
    if (TIF_phase != TIF_PHASE_CHOOSING_RECORDS)
        return ILERROR ((int) TIF_phase, TIF_ERR_WRONG_PHASE);

    //----- allow -1 to specify the CURRENT record

```

```

    if (lRecNum == -1)
        lRecNum = TIF_CurrentRecordNumber;

    //---- make sure that we have a valid record# to work with
    rc = TIFValidateRecord (pstTIF, lRecNum);
    if (rc == SUCCESS)
    {
        //---- mark the 'removed' record as garbage
        TIFX_FLAGS(phFile, lRecNum) |= TIF_IS_GARBAGE;

        //---- if record belongs to an SKG, remove it from it
        rc = TIFremoveFromSKG (phFile, lRecNum, NULL);
    }

    //---- decide how badly we want to log this activity, and log it
    if (rc == SUCCESS)
        loglevel = 80;
    else
        loglevel = ILLOG_MINOR_ERROR;

    TIFlog2ints ( loglevel, "ILTIFRemoveRecord(%ld) ==> rc=%ld",
                  (UINT32) lRecNum, (UINT32) rc );
    return rc;
} //---- ILTIFRemoveRecord

#ifdef USE_ILCR_STUBS

extern INT32 stubFirstItem; // see TIF_ILCR.CPP

int IL_DECL ILCRBegin (ILTR_PTRANSI tr, IL_HINST h, BOOLEAN b)
{
    ILTIFlogsz("ILCRBegin stub called");
    stubFirstItem = TIF_NOTSET;
    return 0;
}

int IL_DECL ILCREnd (ILTR_PTRANSI tr, IL_HINST h)
{
    ILTIFlogsz("ILCREnd stub called"); return 0;
}

#endif

```

```

/*-----
* Name:      TIF.CPP
* Purpose:   This file contains modules that access ILDFX in a convenient way
*           for the generic translator module.  The routines within are
*           responsible for writing and reading the Translator Intermediate
*           File(TIF).
*
*           The TIF is set up like a normal ILDFX file with the addition that
*           the data stored is in the record format described below:
*
*           -----
*           | Record Header
*           |   lRecSize  - LONG - Size of Record
*           |   FieldCount - INT16 - Number of Fields
*           |   lNextOffset - LONG - points just beyond last occupied byte
*           |   Field Value Headers ( 1 to FieldCount of the following)
*           |       fieldOffset - LONG - offset to where field data is
*           |       lFldSize  - LONG - Size of the data
*           |   Field Data -- all packed together
*           |-----
*
*           The first record in the file stores the field list description
*           (see TIF.H).
*
*           Functions (local functions indented):
*
*               FieldListAlloc
*               EnlargeFieldList
*               TIFRecordAddFieldValue
*               IdentifyDistinguishedFields
*               IdentifyDefaultFields
*               LogCodeVersion
*               FirstInit
*               SecondInit
*               TIFInit
*               TIFLoadKStruct
*               WrapUpSetUp
*               PhaseName
*               SetLoadingRange
*               ComputeSourceOutcomeCounts
*               TIFStartNextPhase
*               TIFSaveFanoutMaxima
*               TIFTerminate
*               TIFFreeBuffers
*               TIFInitRecord
*               TIFHowManyField
*               TIFGetFieldName
*               ReflectSourceAttribs
*               TIFDefineOneField
*               TIFLookUpFieldDefinition
*               TIFPutFieldByIndex
*               UseFactoryDefault
*               TIFFillDefaults
*               TIFRetrieveRecord
*               TIFBuildSPT
*               TIFSetRecordNumbers
*               TIFCopyUnmappedFields
*               TIFGetRecord
*               TIFRetrieveFieldByIndex
*               RetrieveFieldValue
*               TIFFieldChanged
*               TIFGetField
*               GetFieldByIndex
*               SelectRecord
*               TIFGetViewField
*               ConsiderSmartMergeItem
*               ConsiderThisItem
*               TIFComputePertinentRecordCount
*               TIFPositionToNextRecord
*               TIFValidateRecord
*               TIFGetOutcome
*               LogILTRFieldList
*
* Author:   Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Author:   David Boothby, Copyright (c) IntelliLink Corp., 1995

```



```

-----*/
#include "iltr.h"
#include "tifsinc2.h"

#define LOG_ERR_AND_EXIT(e1, e2) { rc=ILERROR(e1, e2); goto Exit; }

static int ConsiderThisItem(PSTTIF_TYPE pstTIF, INT32 Item);

static int LogILTRFieldList(ILTR_PTRANSL tr);

static int RetrieveFieldValue ( PSTTIF_TYPE pstTIF,
                                TIF_FIELDLIST_PTR pFL,
                                TIF_RECORD_VALUE_PTR pRecord,
                                INT16 fieldNumber,
                                INT32 IL_DIST *plFieldLength, // OUT
                                ILUT_PBUFFER pField );

static int GetFieldByIndex
( PSTTIF_TYPE pstTIF,
  int fieldnum,
  int nWhich,
  LONG *plFieldLength,
  ILUT_PBUFFER pField );

static int SelectRecord ( PSTTIF_TYPE pstTIF,
                          int fieldnum,
                          int nWhich,
                          int *pnSelect,
                          BOOLEAN *pbMexCurrent );

/*-----
 * Name:      FieldListAlloc
 *-----*/
static int FieldListAlloc ( PSTTIF_TYPE pstTIF,
                            TIF_FIELDLIST_PTR *ppFieldList,
                            IL_HANDLE *phFieldList,
                            INT16 fieldcount )
{
    size_t mysize = sizeof ( TIF_FIELDLIST ) +
                    ( sizeof ( TIF_FIELD_DESC ) * fieldcount );

    //----- Allocate an initial field info buffer
    *ppFieldList = (TIF_FIELDLIST_PTR) IL_ALLOC (mysize, *phFieldList);
    if (*ppFieldList == NULL)
        return ILERROR_L ((INT32) mysize, TIF_ERR_MEM);

    //----- Clear the field info section
    IL_MEMSET (*ppFieldList, 0, mysize);

    for (INT16 i=0; i < fieldcount; i++)
    {
        (*ppFieldList)->pFieldDescs[i].FieldNum = TIF_NOTSET;
        (*ppFieldList)->pFieldDescs[i].RelatedFieldNum = TIF_NOTSET;
    }

    (*ppFieldList)->ActualFieldCount = 0L;
    (*ppFieldList)->AllocatedFieldCount = fieldcount;

    return SUCCESS;
} //---- FieldListAlloc

/*-----
 * Name:      EnlargeFieldList
 *-----*/
static int EnlargeFieldList ( PSTTIF_TYPE pstTIF,
                              TIF_FIELDLIST_PTR *ppFieldList,
                              IL_HANDLE *phFieldList )
{
    INT16 OldCount = (*ppFieldList)->AllocatedFieldCount;
    INT16 NewCount = OldCount + TIF_FIELD_COUNT_INCREMENT;
    size_t NewSize = sizeof ( TIF_FIELDLIST ) +
                    ( sizeof ( TIF_FIELD_DESC ) * NewCount );

```

```

    *ppFieldList = (TIF_FIELDLIST_PTR)
        IL_REALLOC (NewSize, *phFieldList, *ppFieldList);
    if (*ppFieldList == NULL)
        return ILERROR_L ((INT32) NewSize, TIF_ERR_MEM);

    // Clear the new part of the field info section
    IL_MEMSET( &(*ppFieldList)->pFieldDescs[OldCount], 0,
        sizeof(TIF_FIELD_DESC) * TIF_FIELD_COUNT_INCREMENT );

    for (INT16 i=OldCount; i < NewCount; i++)
    {
        (*ppFieldList)->pFieldDescs[i].FieldNum = TIF_NOTSET;
        (*ppFieldList)->pFieldDescs[i].RelatedFieldNum = TIF_NOTSET;
    }

    (*ppFieldList)->AllocatedFieldCount = NewCount;

    return SUCCESS;
} //----- EnlargeFieldList

/*-----
* Name:      TIFRecordAddFieldValue
*
* You can either pass a Field Name, in 2nd arg, or pass a Field Number,
* in 3rd arg.  When passing a Field Number, put NULL in the 2nd arg.
*
* NOTE:  for non-binary fields, the 'lData' arg is usually ignored ... we use
*         IL_STRLEN to compute the length of non-binary 'pData'.  But if
*         'lData' has the TIF_LENGTH_GUARANTEED bit set then we respect lData.
*-----*/
int TIFRecordAddFieldValue ( PSTTIF_TYPE pstTIF,
    TIF_FIELDLIST_PTR pFL,
    IL_PSTR szFldName,          // supply either a field name...
    INT16 fieldnum,            // ...or a field number
    IL_PSTR pData,
    INT32 Length,
    BOOLEAN bMapCharsIfNonBinary,
    ILUT_PBUFFER pRecord )
{
    //----- grab the 'LengthGuaranteed' bit
    BOOLEAN bUseSTRLEN = ((Length & TIF_LENGTH_GUARANTEED) == 0);
    Length &= ~TIF_LENGTH_GUARANTEED;

    INT32 Size;
    int rc;
    ILTR_PTRANS� tr = TIF_tr;
    TIF_RECORD_VALUE_PTR pRecData = (TIF_RECORD_VALUE_PTR) (pRecord->pBuffer);

    if (pData == NULL)
        return TIF_ERR_PDATA_IS_NULL;

    /*-----
    * If field is specified by name, look up fieldnum in TIF Field List.
    *-----*/
    if (IL_STRING_ISNT_NULL(szFldName))
    {
        // Loop through the fields until we find the one we are looking for
        fieldnum = TIF_NOTSET;
        INT16 fieldcount = TIF_FieldCount2(pFL);
        for (INT16 i = 0; i < fieldcount; i++)
            if (IL_STRINGS_EQUAL(szFldName, TIF_FieldName2(pFL, i)))
            {
                fieldnum = i;
                break;
            }

        if (fieldnum == TIF_NOTSET)
            return TIF_ERR_BAD_FLDNAME;
    }

    if (TIF_FieldType2(pFL, fieldnum) == ILX_TYPE_BINARY)
        //----- for binary fields Size=Length (no terminator added)

```

```

    Size = Length;
else
{
    /*-----
    * If field is non-binary, set Length & Size and do character mapping
    *-----*/
    if (bUseSTRELEN)
        Length = IL_STRELEN(pData);

    if (bMapCharsIfNonBinary)
    {
        /*-----
        * Do character mapping, and put result string in ILTR_pTmpBuf.
        * ILTR_pTmpBuf is a reusable buffer (to minimize heap activity)
        *-----*/
        rc = ILMMapChars ( pData,
                           Length,
                           ILTR_szLineTerm, // old line terminator, e.g. \r\n
                           ILTR_EOS_STR,    // new line terminator, e.g. \xFF
                           ILTR_sExportCharMap.buffer,
                           ILTR_MAX_FIELDLNGTH,
                           ILTR_pTmpBuf );
        if (rc != SUCCESS)
            return ILERROR (rc, ILTR_ERR_INTERNAL_ERROR);

        pData = (IL_PSTR) (ILTR_pTmpBuf->pBuffer);
        /*-----
        * Set Length (ALWAYS!!) -- ILMMapChars always null-terminates. For
        * example, if input to ILMMapChars is pData="Hello", Length=2,
        * then output is pData="He" (null-terminated).
        *-----*/
        Length = IL_STRELEN(pData);
    }

    //---- for text fields Size=Length+1 (terminator added)
    Size = Length+1;
}

if ( (TIF_FieldOffset(pRecData, fieldnum) != TIF_NOTSET)
    && (TIF_FieldLength(pRecData, fieldnum) >= Size) )
{
    /*-----
    * New value will fit where old value was stored. Note that we lose
    * track of the full available size. For example, if you first store
    * an 80-byte field value, then replace that with a 10-byte field value,
    * then yet again replace that with a 20-byte value, that last storage
    * event will cause us to give up on the original space, cuz it doesn't
    * look big enough any more. We then assign a new offset for the field,
    * at the end of the data block for the current record.
    * This may seem a big ugly, but it keeps things simple, and handles
    * same-size replacements effectively.
    * We could keep track of wasted space, so someday we could do garbage
    * collection when wasted space in a record exceeds some threshold.
    *-----*/
    ; // don't bother counting wasted bytes
}
else
{
    /*-----
    * need to append this field value to end of this record's data block.
    * if there was a previous field value, the old space is now wasted.
    *-----*/

    //---- Get current size of record buffer
    INT32 RecordSize = TIFREC_SIZE(pRecData);

    // New field value will be appended to end of record; store offset
    TIF_FieldOffset(pRecData, fieldnum) = RecordSize;

    //---- Calculate how big record buffer needs to be...
    RecordSize += Size;

    //---- Make sure buffer is big enough; expand it if necessary
    rc = ILUT_GetBuffer (pRecord, RecordSize);
    if (rc != SUCCESS)

```

```

        return ILERROR_L (RecordSize, TIF_ERR_MEM);

    //---- expansion may have moved the buffer, so re-set local pointer
    pRecData = (TIF_RECORD_VALUE_PTR) (pRecord->pBuffer);

    //---- Store expanded record size
    TIFREC_SIZE(pRecData) = RecordSize;
}

// Copy new field data into record data block at the selected offset
IL_PSTR pWhere;

pWhere = TIF_FieldData(pRecData, fieldnum);
IL_MEMCPY(pWhere, pData, (size_t) Length);

// NULL-terminate Text Fields
if (Size == Length+1)
    pWhere[Length] = 0;

// Save the new size for this field
TIF_FieldLength(pRecData, fieldnum) = Size;

return SUCCESS;
} //---- TIFRecordAddFieldValue

/*-----
 * Name:      IdentifyDistinguishedFields
 * Find all distinguished fields; record the index of each
 * such field in the *pstTIF structure.
 * This applies to Start&End Date&Time, to ID fields,
 * and to the "View" Field (whose value is used to identify
 * a record in the logfile).
 *
 * When a Start or End Date or Time Field is a duration, relative to
 * another field, things are a little un-obvious, as per the following
 * example:
 *
 *   Support Field 7 is a StartTime; Type=TIME, associatedField=NONE,
 *                               with isStartDateOrTime=TRUE.
 *
 *   and Field 10 is a Duration; Type=NUMBER, associatedField=7,
 *                               with isEndDateOrTime=TRUE.
 *
 *   Then we know that Field 10 is an EndTime field because it has
 *   isEndDateOrTime=TRUE and the type of its associated field is TIME.
 *-----*/
static int IdentifyDistinguishedFields ( PSTTIF_TYPE pstTIF )
{
    ILTR_PTRANS tr = TIF_tr;
    INT16 fieldcount = TIF_FieldCount(pstTIF);
    for (INT16 fieldnum = 0; fieldnum < fieldcount; fieldnum++)
    {
        IL_PSTR fieldname = TIF_FieldName(pstTIF, fieldnum);
        TIF_FIELD_DESC_PTR pFieldDesc = &TIF_FieldDesc(pstTIF, fieldnum);

        // Kludgy way of setting the Source and Target ID flags
        if (IL_STRINGS_EQUAL(fieldname, TIF_SOURCEID_FIELDNAME))
        {
            if (TIF_SourceIDFieldNum == TIF_NOTSET)
            {
                TIF_SourceIDFieldNum = fieldnum;
                TIF_bSyncUsingSourceIDs = TRUE; // unless overridden later
                continue;
            }
            else
            {
                TIFlogszulul( "Error setting SourceIDFieldNum to %#ld (already set to %#ld)",
                    (UINT32) fieldnum, (UINT32) TIF_SourceIDFieldNum );
                return ILERROR_L (TIF_SourceIDFieldNum, TIF_ERR_DUP_DISTING_FIELD);
            }
        }
        else if (IL_STRINGS_EQUAL(fieldname, TIF_TARGETID_FIELDNAME))
    }

```

```

{
    if (TIF_TargetIDFieldNum == TIF_NOTSET)
    {
        TIF_TargetIDFieldNum = fieldnum;
        TIF_bSyncUsingTargetIDs = TRUE; // unless overridden later
        continue;
    }
    else
    {
        TIFlogszulul( "Error setting TargetIDFieldNum to %ld (already set to %ld)",
                      (UINT32) fieldnum, (UINT32)TIF_TargetIDFieldNum );
        return ILERROR_L (TIF_TargetIDFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
}

else if (IL_STRINGS_EQUAL(fieldname, ILTR_REP_BASIC))
{
    if (TIF_RepBasicFieldNum == TIF_NOTSET)
    {
        TIF_RepBasicFieldNum = fieldnum;
        continue;
    }
    else
    {
        TIFlogszulul( "Error setting RepBasicFieldNum to %ld (already set to %ld)",
                      (UINT32) fieldnum, (UINT32)TIF_RepBasicFieldNum );
        return ILERROR_L (TIF_RepBasicFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
}

else if (IL_STRINGS_EQUAL(fieldname, ILTR_REP_XDATE))
{
    if (TIF_RepExclFieldNum == TIF_NOTSET)
    {
        TIF_RepExclFieldNum = fieldnum;
        continue;
    }
    else
    {
        TIFlogszulul( "Error setting RepExclFieldNum to %ld (already set to %ld)",
                      (UINT32) fieldnum, (UINT32)TIF_RepExclFieldNum );
        return ILERROR_L (TIF_RepExclFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
}

else if (IL_STRINGS_EQUAL(fieldname, ILTR_FLD_DELTA))
{
    if (TIF_TargetDeltaFieldNum == TIF_NOTSET)
    {
        TIF_TargetDeltaFieldNum = fieldnum;
        continue;
    }
    else
    {
        TIFlogszulul( "Error setting TargetDeltaFieldNum to %ld (already set to
%ld)",
                      (UINT32) fieldnum, (UINT32)TIF_TargetDeltaFieldNum );
        return ILERROR_L (TIF_TargetDeltaFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
}

else if (IL_STRINGS_EQUAL(fieldname, TIF_USFN_PREFIX ILTR_FLD_DELTA))
{
    if (TIF_SourceDeltaFieldNum == TIF_NOTSET)
    {
        TIF_SourceDeltaFieldNum = fieldnum;
        continue;
    }
    else
    {
        TIFlogszulul( "Error setting SourceDeltaFieldNum to %ld (already set to
%ld)",
                      (UINT32) fieldnum, (UINT32)TIF_SourceDeltaFieldNum );
        return ILERROR_L (TIF_SourceDeltaFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
}

```

```

    }

    /*-----
    * Unlike the preceding SPECIAL fields, which may be "distinguished"
    * w/o being MAPPED, all of the remaining "distinguished field" cases
    * require that the field be MAPPED. So don't consider any unmapped
    * fields as candidates for being distinguished, beyond this point.
    *-----*/
    else if (pFieldDesc->ExtraAttributes & TIFEA_ISNT_MAPPED)
        continue;

    //----- If it is a view field, save the View index
    if (pFieldDesc->FieldAttributes & ILTB_ATT_VIEW)
    {
        if (TIF_ViewFieldNum == 0)
            TIF_ViewFieldNum = fieldnum;
        else
        {
            TIFlogszulul( "Error setting ViewFieldNum to %ld (already set to %ld)",
                (UINT32) fieldnum, (UINT32) TIF_ViewFieldNum );
            return ILERROR_L (TIF_ViewFieldNum, TIF_ERR_DUP_DISTING_FIELD);
        }
    }

    /*-----
    * For non-relative fields, rootField is same as field. But for
    * relative fields rootField is the associated field.
    *-----*/
    IL_PSTR rootfieldname = TIF_RelatedFieldName(pstTIF, fieldnum);
    INT16 rootfieldnum;
    if (IL_STRING_IS_NULL(rootfieldname))
    {
        rootfieldnum = fieldnum;
        TIF_RelatedFieldNum(pstTIF, fieldnum) = TIF_NOTSET;
    }
    else
    {
        rootfieldnum = TIF_NOTSET;
        for (int i = 0; i < fieldcount; i++)
            if (IL_STRINGS_EQUAL(TIF_FieldName(pstTIF, i), rootfieldname))
            {
                rootfieldnum = i; break;
            }

        TIF_RelatedFieldNum(pstTIF, fieldnum) = rootfieldnum;

        if (rootfieldnum == TIF_NOTSET)
        {
            /*-----
            * We get here if the root field that the distinguished field
            * is relative to is not mapped.
            *-----*/
            continue; //do not return TIF_ERR_CANT_FIND_ROOT_FIELD;
        }
    }

    int rootfieldtype;
    rootfieldtype = TIF_FieldType(pstTIF, rootfieldnum);

    if (rootfieldtype == ILX_TYPE_DATE)
    {
        if (pFieldDesc->FieldAttributes & ILTB_ATT_STARTDATETIME)
        {
            if (TIF_StartDateFieldNum == TIF_NOTSET)
                TIF_StartDateFieldNum = fieldnum;
            else
            {
                TIFlogszulul( "Error setting StartDateFieldNum to %ld (already set to
%ld)",
                    (UINT32) fieldnum, (UINT32) TIF_StartDateFieldNum );
                return ILERROR_L (TIF_StartDateFieldNum, TIF_ERR_DUP_DISTING_FIELD);
            }
        }
        else if (pFieldDesc->FieldAttributes & ILTB_ATT_ENDDATETIME)
        {
            if (TIF_EndDateFieldNum == TIF_NOTSET)
                TIF_EndDateFieldNum = fieldnum;
            else
            {

```

```

        TIFlogszulul( "Error setting EndDateFieldNum to %ld (already set to
%ld)",
                    (UINT32) fieldnum, (UINT32)TIF_EndDateFieldNum );
        return ILERROR_L (TIF_EndDateFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
} else if (pFieldDesc->FieldAttributes & ILTB_ATT_ALARMRELATED)
{
    if (TIF_AlarmDateFieldNum == TIF_NOTSET)
        TIF_AlarmDateFieldNum = fieldnum;
    else
    {
        TIFlogszulul( "Error setting AlarmDateFieldNum to %ld (already set to
%ld)",
                    (UINT32) fieldnum, (UINT32)TIF_AlarmDateFieldNum );
        return ILERROR_L (TIF_AlarmDateFieldNum, TIF_ERR_DUP_DISTING_FIELD);
    }
}

} //---- if (rootfieldtype == ILX_TYPE_DATE)

else if (rootfieldtype == ILX_TYPE_TIME)
{
    if (pFieldDesc->FieldAttributes & ILTB_ATT_STARTDATETIME)
    {
        if (TIF_StartTimeFieldNum == TIF_NOTSET)
            TIF_StartTimeFieldNum = fieldnum;
        else
        {
            TIFlogszulul( "Error setting StartTimeFieldNum to %ld (already set to
%ld)",
                        (UINT32) fieldnum, (UINT32)TIF_StartTimeFieldNum );
            return ILERROR_L (TIF_StartTimeFieldNum, TIF_ERR_DUP_DISTING_FIELD);
        }
    }
    else if (pFieldDesc->FieldAttributes & ILTB_ATT_ENDDATETIME)
    {
        if (TIF_EndTimeFieldNum == TIF_NOTSET)
            TIF_EndTimeFieldNum = fieldnum;
        else
        {
            TIFlogszulul( "Error setting EndTimeFieldNum to %ld (already set to
%ld)",
                        (UINT32) fieldnum, (UINT32)TIF_EndTimeFieldNum );
            return ILERROR_L (TIF_EndTimeFieldNum, TIF_ERR_DUP_DISTING_FIELD);
        }
    }
    else if (pFieldDesc->FieldAttributes & ILTB_ATT_ALARMRELATED)
    {
        if (TIF_AlarmTimeFieldNum == TIF_NOTSET)
            TIF_AlarmTimeFieldNum = fieldnum;
        else
        {
            TIFlogszulul( "Error setting AlarmTimeFieldNum to %ld (already set to
%ld)",
                        (UINT32) fieldnum, (UINT32)TIF_AlarmTimeFieldNum );
            return ILERROR_L (TIF_AlarmTimeFieldNum, TIF_ERR_DUP_DISTING_FIELD);
        }
    }
}

} //---- else if (rootfieldtype == ILX_TYPE_TIME)

else if (rootfieldtype == ILX_TYPE_BOOL)
{
    if (pFieldDesc->FieldAttributes & ILTB_ATT_ALARMRELATED)
    {
        if (TIF_AlarmFlagFieldNum == TIF_NOTSET)
            TIF_AlarmFlagFieldNum = fieldnum;
        else
        {
            TIFlogszulul( "Error setting AlarmFlagFieldNum to %ld (already set to
%ld)",
                        (UINT32) fieldnum, (UINT32)TIF_AlarmFlagFieldNum );
            return ILERROR_L (TIF_AlarmFlagFieldNum, TIF_ERR_DUP_DISTING_FIELD);
        }
    }
}

```

```

    }

    if ( ILTR_nFunction == ILTR_TODO
        && (pFieldDesc->FieldAttributes & ILTB_ATT_DONEFLAG) )
    {
        if (TIF_DoneFlagFieldNum == TIF_NOTSET)
            TIF_DoneFlagFieldNum = fieldnum;
        else
        {
            TIFLogszulul( "Error setting DoneFlagFieldNum to #%ld (already set to
                #%ld)",
                        (UINT32) fieldnum, (UINT32) TIF_DoneFlagFieldNum );
            return ILERROR_L (TIF_DoneFlagFieldNum, TIF_ERR_DUP_DISTING_FIELD);
        }
    }

    } //---- else if (rootfieldtype == ILX_TYPE_BOOL)

} //---- end for loop

//--- verify that _subType fields were found (by TIFDefineOneField)
if (TIF_SubTypeFieldNum == TIF_NOTSET)
    return ILERROR(0, TIF_ERR_NO_SUBTYPE_FIELD);

if ( (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
    && (TIF_SourceSubTypeFieldNum == TIF_NOTSET) )
    return ILERROR(1, TIF_ERR_NO_SUBTYPE_FIELD);

//--- determine whether translators are FastSync-capable
TIF_bSourceIsFastSyncAble = ( TIF_SourceIDFieldNum != TIF_NOTSET
    && TIF_SourceDeltaFieldNum != TIF_NOTSET );

TIF_bTargetIsFastSyncAble = ( TIF_TargetIDFieldNum != TIF_NOTSET
    && TIF_TargetDeltaFieldNum != TIF_NOTSET );

return SUCCESS;

} //---- IdentifyDistinguishedFields

/*-----
* Name:      IdentifyDefaultFields
* For all fields that get their default values from other fields, set
* the TIF_FieldToGetDefaultValueFrom member of the field descriptor.
*
* NOTE:  this function pays no attention to the "Value Required" field
*        attribute.  But having Default Value rules set up is all for
*        nought if the "Value Required" attribute isn't set.  The
*        TIFFillDefaults function will only supply default values for
*        fields that lack a value and have "Value Required" attribute.
*-----*/
static int IdentifyDefaultFields (ILTR_PTRANSL tr, PSTTIF_TYPE pstTIF)
{
    int fieldcount = TIF_FieldCount(pstTIF);
    for (int fieldnum = 0; fieldnum < fieldcount; fieldnum++)
    {
        /*-----
        * Initial assumption, for every field, is that it does NOT get a
        * default value from anywhere.
        *-----*/
        TIF_FieldToGetDefaultValueFrom(pstTIF, fieldnum) = TIF_NOTSET;

        IL_PSTR szDefault = TIF_FieldDefaultValue(pstTIF, fieldnum);
        if (IL_STRING_IS_NULL(szDefault) || (szDefault[0] != '\\'))
        {
            /*-----
            * If default value is NULLSTRING, or doesn't start with BACKSLASH,
            * nothing needs doing.
            *-----*/
            continue;

            //--- get ZERO-based index into Field Map.
            int nInFieldMap; nInFieldMap = atoi(&szDefault[1]) - 1;

            //--- get Internal Name for the Field to get default value from
            char FieldName[ILTR_MAX_FLDNAME+1];

```



```

int rc; rc = ILFldGetIntName (tr, nInFieldMap, FieldName);
if (rc != SUCCESS)
{
    TIFlogszulul( "Field #%ld used as default by field #%ld does not exist",
                  (UINT32) nInFieldMap+1, (UINT32) fieldnum+1 );

    /*-----
    * One way to get here is if field F in a fieldlist of N fields
    * asks for default to come from field G, where G > N.
    *-----*/
    // return ILERROR(nInFieldMap, rc); // uncomment to find bugs!!!

    IL_MAKE_STRING_NULL (szDefault);
    continue;
}

TIFlogsz3("Default values for '%s' will come from '%s'",
          TIF_FieldName(pstTIF, fieldnum), FieldName);

// Find specified field in TIF Field List
int nInTIF; nInTIF = TIF_NOTSET;
int i;
for (i = 0; i < fieldcount; i++)
{
    if (IL_STRINGS_EQUAL(FieldName, TIF_FieldName(pstTIF, i)))
    { nInTIF = i; break; }
}

if (nInTIF == TIF_NOTSET)
{
    /*-----
    * Perhaps the field we want to get the default from is unmapped.
    * Log a warning, set default value to NULLSTRING and go on.
    *-----*/
    TIFlogszsz("WARNING: field %s is not in TIF's fieldlist", FieldName);
    IL_MAKE_STRING_NULL (szDefault);
}

TIF_FieldToGetDefaultValueFrom(pstTIF, fieldnum) = (INT16) nInTIF;
} //---- end for loop

return SUCCESS;
} //---- IdentifyDefaultFields

#ifdef ILWIN
#ifdef WIN32
#include <winver.h>
#else
#include <ver.h>
#endif

/*-----
* Name: LogCodeVersion (called from FirstInit)
*-----*/
static int LogCodeVersion(PSTTIF_TYPE pstTIF)
{
    char szFile[MAX_PATH];
    DWORD len32, x32;
    UINT len16;
    BOOL b;
    IL_PSTR p;
    IL_PANY p2;

    len32 = GetModuleFileName (TIF_DLL_InstanceHandle, szFile, MAX_PATH);
    if (len32 == 0)
        return ILERROR (0, TIF_ERR_ABNORMAL);

    len32 = GetFileVersionInfoSize (szFile, &x32);
    if (len32 == 0)
        return ILERROR_S (szFile, TIF_ERR_ABNORMAL);
}

```

```

    p = new char[len32];
    if (p == NULL)
        return ILERROR_L ((INT32) len32, TIF_ERR_MEM);

    b = GetFileVersionInfo (szFile, x32, len32, p);
    if (b == 0)
    {
        delete p;
        return ILERROR_S (szFile, TIF_ERR_ABNORMAL);
    }

    char szQuery[40];
    IL_STRCPY (szQuery, "\\StringFileInfo\\040904E4\\Comments");

    p2 = NULL;
    len16 = 0;
    b = VerQueryValue (p, szQuery, &p2, &len16);
    if (b == 0 || p2 == NULL || len16 == 0)
    {
        delete p;
        return ILERROR_S (szFile, TIF_ERR_ABNORMAL);
    }

    TIFlogsz3 ("\r\n ----- Running '%s' %s\r\n", szFile, (IL_PSTR) p2);
    delete p;
    return SUCCESS;

} //---- LogCodeVersion

#endif //---- #ifdef ILWIN

/*-----
 * Name:      FirstInit
 * Called from TIFInit on a first-time init
 *-----*/
static int FirstInit ( ILTR_PTRANSL tr,
                      PSTTIF_TYPE pstTIF,
                      int FieldCount )
{
    int rc, rc2;
    IL_FILEINFO info;          // struct needed to remove file

    if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 20))
    {
        //---- log some of the basic parameters for current translation operation
        char sz[300];

        TIFlogsz3 ("%s ==> %s", ILTR_szSrcTrans, ILTR_szTarTrans);
        TIFlogsz ("-----");
        TIFlogsz3 ("SOURCE: %s %s", ILTR_szAltApp, ILTR_szAltSect);
        IL_SPRINTF(sz, "TARGET: %s %s (%s)", ILTR_szAppName, ILTR_szSectName,
ILTR_szSectCode);
        TIFlogsz(sz);
        TIFlogszsz("      %s", ILTR_szAppFile);

        IL_SPRINTF(sz,
"src=%d.%d, tar=%d.%d, func=%d, map#%d, phase=%d, ILLOGverbosity=%d",
                ILTR_nSourceID, ILTR_nSrcSection,
                ILTR_nTargetID, ILTR_nTarSection,
                ILTR_nFunction, ILTR_nMapID,
                ILTR_phase, TIFLOG->nVerbosity);
        TIFlogsz(sz);
        IL_SPRINTF(sz,
"v=%d, sync=%d, UpdOpt=%d, policy=0x%lx, ILTR_Flags=0x%lx, sizeof(tr)=%d",
                ILTR_version, ILTR_nSynchronize,
                ILTR_nUpdOpt, ILTR_CRPolicy, ILTR_Flags, sizeof(ILTR_TRANSL));
        TIFlogsz(sz);

        TIFlogszul3 ( "TIFVersion=0x%lx, SourceSST=%ld, TargetSST=%ld",
                TIF_CURRENT_FILE_FORMAT_VERSION,
                (UINT32) ILTR_SourceSST, (UINT32) ILTR_TargetSST );

        IL_PSTR psz;

```

```

switch (ILTR_nSynchronize)
{
    case ILXTR_SYNC_NO:          psz = "SmartMerge";          break;
    case ILXTR_SYNC_STANDARD:    psz = "Incremental Synchronization"; break;
    case ILXTR_SYNC_FROM_SCRATCH: psz = "Synchronization from Scratch"; break;

    default: IL_SPRINTF (sz, "Sync Type %d", ILTR_nSynchronize); psz = sz;
}

TIFlogszsz ("\r\n ----- Requested Service is %s", psz);

#ifdef ILWIN
    rc = LogCodeVersion(pstTIF); // rc is ignored
#else
    TIFlogsz(" ");
#endif
}

if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 85))
{
    rc = LogILTRFieldList(tr);
    if (rc != SUCCESS)
        return ILERROR(rc, TIF_ERR_INIT_FAILURE);
}

TIF_phase = TIF_PHASE_SETTING_THINGS_UP;
TIF_PreviousILTRPhase = 0;
TIF_TotalRecordCount = TIF_FIRST_ITEMNO;

//---- Zero out the Distinguished Field indices
TIF_AlarmFlagFieldNum      = TIF_NOTSET;
TIF_AlarmTimeFieldNum      = TIF_NOTSET;
TIF_AlarmDateFieldNum      = TIF_NOTSET;
TIF_DoneFlagFieldNum       = TIF_NOTSET;
TIF_StartDateFieldNum      = TIF_NOTSET;
TIF_EndDateFieldNum        = TIF_NOTSET;
TIF_StartTimeFieldNum      = TIF_NOTSET;
TIF_EndTimeFieldNum        = TIF_NOTSET;
TIF_ViewFieldNum           = TIF_NOTSET;
TIF_SourceIDFieldNum       = TIF_NOTSET;
TIF_TargetIDFieldNum       = TIF_NOTSET;
TIF_SourceDeltaFieldNum    = TIF_NOTSET;
TIF_TargetDeltaFieldNum    = TIF_NOTSET;
TIF_RepBasicFieldNum       = TIF_NOTSET;
TIF_RepExclFieldNum        = TIF_NOTSET;
TIF_SubTypeFieldNum        = TIF_NOTSET;
TIF_SourceSubTypeFieldNum  = TIF_NOTSET;
TIF_KeyDateFieldNum        = TIF_NOTSET;

/*-----
 * By default use Field Zero as the "View Field", which is used
 * to tell the user what field we're talking about, in conflict
 * resolution dialog. Typically a field in the field list will
 * have the ViewField attribute set, in which case the default
 * value of TIF_ViewField will be overridden.
 *-----*/
TIF_ViewFieldNum = 0;

/*-----
 * Set policy for Appointment Overlap Detection.
 * This should be set to FALSE if section is
 * not appointments, or for Synchronization, or if desired
 * conflict resolution policy is to not bother doing overlap
 * detection.
 *-----*/
TIF_bCheckForOverlap =
    ( (ILTR_nFunction == ILTR_APPT)
      && ((TIF_CRPolicy & ILXTR_CR_POLICY_DETECT_APPT_OVERLAPS) != 0) );

//---- Tell TIF whether or not to exclude completed todos from sync
if (ILTR_nSynchronize && ILTR_nFunction == ILTR_TODO)
    TIF_bDontSyncDoneTodos = (ILTR_nSchOpt == ILTB_TODO_NOTDONE);
else
    TIF_bDontSyncDoneTodos = FALSE;

```

```

// Allocate the field info structures
rc = FieldListAlloc ( pstTIF, &TIF_pFieldList,
                      &TIF_hFieldList, FieldCount );

if (rc == SUCCESS)
    rc = FieldListAlloc ( pstTIF, &TIF_pSourceFieldList,
                          &TIF_hSourceFieldList, FieldCount );
if (rc != SUCCESS)
    return ILERROR(rc, TIF_ERR_INIT_FAILURE);

//--- get pseudo-fieldnames to use in ILCR display
rc2 = ILSTMakeString ( &hXlatorInst, TIF_STR_REPSTART_PSEUDO_FLDNAME,
                       TIF_szRepStartPseudoFldName, ILTR_MAX_FLDNAME );

rc2 = ILSTMakeString ( &hXlatorInst, TIF_STR_REPSTOP_PSEUDO_FLDNAME,
                       TIF_szRepStopPseudoFldName, ILTR_MAX_FLDNAME );

rc2 = ILSTMakeString ( &hXlatorInst, TIF_STR_REPEX_PSEUDO_FLDNAME,
                       TIF_szRepExPseudoFldName, ILTR_MAX_FLDNAME );

rc2 = ILSTMakeString ( &hXlatorInst, TIF_STR_TRUE,
                       TIF_szTrue, sizeof(TIF_szTrue) );

rc2 = ILSTMakeString ( &hXlatorInst, TIF_STR_FALSE,
                       TIF_szFalse, sizeof(TIF_szFalse) );

TIF_bReconcileRepBasic = TRUE;
TIF_bReconcileExclusions = TRUE;

// generate unique filename for TIF file
IL_PSTR p; p = ILTR_TempFileName (ILTR_nFunction, TIF_szWorkFile);
if (p == NULL)
    return ILERROR(0, TIF_ERR_INIT_FAILURE);

//---- for ILX_V4 mode put workfile name into "tr" structure too
if (ILTR_phase != ILTR_PHASE_ILX_V3_MODE)
    IL_STRCPY (ILTR_szWorkFile, TIF_szWorkFile);

rc = ILDFX_CreateFile (TIF_szWorkFile,
                       200, // initial size of index
                       TIF_EXDATA_PER_RECORD );
if (rc != SUCCESS)
    return ILERROR(rc, TIF_ERR_INIT_FAILURE);

//--- For Appts & Todos, define TIF-internal instance list field
if ((ILTR_nFunction == ILTR_APPT) || (ILTR_nFunction == ILTR_TODO))
{
    INT32 extraAttributes=TIFEA_ISNT_MAPPED;
    rc = TIFDefineOneField ( pstTIF,
                            TIF_pFieldList, // put in primary list
                            TIF_INST_LIST_FIELD, // fieldname
                            0, // size (unlimited)
                            ILX_TYPE_BINARY, // type (binary)
                            "", // format
                            "", // relative field
                            ILTB_ATT_NO_RECONCILE | ILTB_ATT_HIDDEN_FIELD,
                            &extraAttributes,
                            TRUE, // positive (irrelevant)
                            "", // default value
                            TIF_NOTSET ); // ILTR fieldnum -- none!!

    if (rc != SUCCESS)
    {
        IL_REMOVE (TIF_szWorkFile, info, rc2);
        return ILERROR(rc, TIF_ERR_INIT_FAILURE);
    }
}

// Open the workfile for writing
rc = ILDFX_OpenFile (TIF_szWorkFile, TIF_hFile, ILDFX_MODE_UPDATE);
if (rc == SUCCESS)
{
    TIF_bWorkFileIsOpen = TRUE;
    TIF_FileFormatVersion = TIF_CURRENT_FILE_FORMAT_VERSION;
}

```

```

/*-----
 * Supply EXDATA values as eye-catchers and bug-catchers
 *-----*/
INT32 exdata[TIF_EXDATA_PER_RECORD];
int i;

for (i=0; i < TIF_EXDATA_PER_RECORD; i++)
    exdata[i] = 0x7FFFFFF0 + (16 * i) + TIF_RECORD_TWO; // eye-catchers

// Write out the TIF_KSTRUCT as Record TWO, to versionStamp the file
rc = ILDFX_AddRecord ( TIF_hFile,
                      TIF_RECORD_TWO,
                      &((*pstTIF)->K),
                      (INT32) sizeof(TIF_KSTRUCT),
                      exdata );
}

else
{
    IL_REMOVE (TIF_szWorkFile, info, rc2);
    return ILERROR(rc, TIF_ERR_INIT_FAILURE);
}

return rc;
} //---- FirstInit

/*-----
 * Name:          SecondInit
 * Called from TIFInit on a re-init. This is done for ILX_V4 mode
 * only, and only when a 16-bit translator is running under a
 * 32-bit engine.
 *-----*/
static int SecondInit (ILTR_PTRANS� tr, PSTTIF_TYPE pstTIF)
{
    int rc, rc2;
    INT32 len;
    ILDFX_PHNDL phFile = TIF_hFile;

    // Copy workfile name from "tr" structure into TIF structure
    IL_STRCPY (TIF_szWorkFile, ILTR_szWorkFile);

    // Open the workfile for reading & writing
    rc = ILDFX_OpenFile (TIF_szWorkFile, phFile, ILDFX_MODE_UPDATE);
    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_INIT_FAILURE);

    TIF_bWorkFileIsOpen = TRUE;
    TIF_PreviousILTRPhase = ILTR_phase;

    rc = ILDFX_GetRecordCount (phFile, &TIF_TotalRecordCount);
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT(rc, TIF_ERR_INIT_FAILURE);

    /*-----
     * Target Field List comes from Record ZERO of TIF File
     *-----*/
    rc = ILDFX_GetRecordLength (phFile, TIF_RECORD_ZERO, &len);
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT(rc, TIF_ERR_INIT_FAILURE);

    TIF_pFieldList = (TIF_FIELDLIST_PTR) IL_ALLOC (len, TIF_hFieldList);
    if (TIF_pFieldList == NULL)
        LOG_ERR_AND_EXIT(0, TIF_ERR_MEM);

    rc = ILDFX_GetRecord ( phFile,
                          TIF_RECORD_ZERO,
                          NULL,
                          TIF_pFieldList,
                          len,
                          NULL ); //-- no ExData for this record
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT(rc, TIF_ERR_INIT_FAILURE);
}

```

```

/*-----
 * Source Field List comes from Record ONE of TIF File
 *-----*/
rc = ILDFX_GetRecordLength (phFile, TIF_RECORD_ONE, &len);
if (rc != SUCCESS)
    LOG_ERR_AND_EXIT(rc, TIF_ERR_INIT_FAILURE);

TIF_pSourceFieldList = (TIF_FIELDLIST_PTR)
    IL_ALLOC (len, TIF_hSourceFieldList);
if (TIF_pSourceFieldList == NULL)
    LOG_ERR_AND_EXIT(0, TIF_ERR_MEM);

rc = ILDFX_GetRecord ( phFile,
    TIF_RECORD_ONE,
    NULL,
    TIF_pSourceFieldList,
    len,
    NULL ); //-- no ExData for this record
if (rc != SUCCESS)
    LOG_ERR_AND_EXIT(rc, TIF_ERR_INIT_FAILURE);

/*-----
 * TIF_KSTRUCT (misc params) comes from TIF_RECORD_TWO of TIF File
 *-----*/
rc = TIFLoadKStruct (phFile, &((*pstTIF)->K));
if (rc == SUCCESS)
    return SUCCESS;

Exit:

//------ note:  caller must free field lists if they are allocated

rc2 = ILDFX_CloseFile (phFile, ILDFX_DONT_UPDATE);
return rc;

} //------ SecondInit

/*-----
 * Name:          TIFInit
 * Called from:  ILTIFInit and ILTIFReopenFile
 * Input:        FieldCount: Number of fields, or -1 for RE-INIT
 *
 * NOTE:         Re-init is done in ILX_V4 mode operation only,
 *               and only when a 16-bit translator running under
 *               a 32-bit engine calls ILTIFReopenFile.
 *-----*/
int TIFInit ( PSTTIF_TYPE pstTIF,
    ILTR_PTRANSL tr,
    int FieldCount )
{
    IL_HANDLE hstTIF;    // Temp handle to global memory
    int rc;
    BOOLEAN bReInit = (FieldCount == -1);
    BOOLEAN bLogFileOpened = FALSE;

    // Allocate an initial global info record
    *pstTIF = (ILT_PTIF) IL_ALLOC(sizeof(ILT_TIF), hstTIF);
    if (*pstTIF == NULL)
        return ILERROR_L ((INT32) sizeof(ILT_TIF), TIF_ERR_MEM);

    IL_MEMSET (*pstTIF, 0, sizeof(ILT_TIF));

    //------ begin Debug Log
    if (bReInit || (ILTR_Flags & ILTR_FLAG_APPEND_TO_LOGS))
        rc = ILLOG_resume("tif.log", ILTR_hLog, &TIFLOG);
    else
        rc = ILLOG_begin("tif.log", ILTR_hLog, &TIFLOG);

    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT(rc, TIF_ERR_INIT_FAILURE);

    ILTIFLOG = TIFLOG;                // for ILTIF-level logging
    ILDFXLOG(TIF_hFile) = TIFLOG;    // to allow debug-logging from ILDFX handle
    bLogFileOpened = TRUE;

```

```

TIFlogszul ("TIFInit(%ld)", (UINT32) FieldCount);

TIF_hstTIF = hstTIF; // handle to self
TIF_pILTIF = ILTR_pILTIF;
TIF_tr = tr; // old granddad
TIF_nSynchronize = ILTR_nSynchronize;
TIF_RecordHasBeenPutSinceLastGet = FALSE;
TIF_CurrentRecordNumber = TIF_POSITION_BELOW_BOTTOM;
TIF_CurrentRecordOutcome = 0;
TIF_PertinentRecordCount = 0L;
TIF_bWorkFileIsOpen = FALSE;
TIF_bNeedPriming = TRUE;
TIF_bPleaseSaveFanoutMaxima = FALSE;

//---- set TIF Reconciliation Option (for SmartMerge AND Synchronization)
TIF_ReconciliationOption = ILTR_nUpdOpt;

/*-----
 * Copy and adjust synchronization outcomes table. The table is
 * copied from STATIC storage into STACK storage, to allow for
 * conflict-free use by multiple concurrent users and to make sure
 * the adjusted table is common to all execution loci, including
 * various Macintosh code resources.
 *-----*/
if (ILTR_nSynchronize)
{
    rc = TIFTableCopyAndAdjust(pstTIF);
    if (rc != SUCCESS)
        return ILERROR (rc, rc);
}

//---- clear FastSync flags. But TIFStartNextPhase may set them...
TIF_bFastSyncLoad = FALSE;
TIF_bFastSyncUnload = FALSE;
TIF_bFastSyncSourceLoad = FALSE;
TIF_bFastSyncTargetLoad = FALSE;

//---- turn off USER-VISIBLE logging. This is turned on for UNLOADING
TIF_bCurrentlyLoggingAndCountingRecords = FALSE;

//---- don't count outcomes except during 'TIFComputePertinentRecordCount'
TIF_bCountingOutcomes = FALSE;

/*-----
 * Set TIF_CRPolicy. Don't try to get value from ILTR if we're running
 * under an old version app that didn't know about this...
 *-----*/
if (ILTR_VERSION_IS_AT_LEAST(8) && (ILTR_CRPolicy != 0))
    TIF_CRPolicy = ILTR_CRPolicy;
else
    TIF_CRPolicy = ILXTR_CR_POLICY_DEFAULT;

/*-----
 * Initialize reusable buffers used for Records and Fields
 *-----*/
rc = ILUT_InitBuffer ( &TIF_CurrentRecord, 0,
                      TIF_BUF_INC, TIF_MAX_RECORD_SIZE+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &TIF_OriginalRecord, 0,
                          TIF_BUF_INC, TIF_MAX_RECORD_SIZE+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &TIF_FirstRecord, 0,
                          TIF_BUF_INC, TIF_MAX_RECORD_SIZE+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &TIF_SecondRecord, 0,
                          TIF_BUF_INC, TIF_MAX_RECORD_SIZE+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &TIF_SourceRecord, 0,
                          TIF_BUF_INC, TIF_MAX_RECORD_SIZE+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &TIF_CurrentField, TIF_INITIAL_FIELD_BUF_SIZE,
                          TIF_BUF_INC, ILTR_MAX_FIELDLLENGTH+1 );
if (rc == SUCCESS)
    rc = ILUT_InitBuffer ( &TIF_OriginalField, TIF_INITIAL_FIELD_BUF_SIZE,

```

```

        TIF_BUF_INC, ILTR_MAX_FIELDLLENGTH+1 );
    if (rc == SUCCESS)
        rc = ILUT_InitBuffer ( &TIF_ExtraField, TIF_INITIAL_FIELD_BUF_SIZE,
                               TIF_BUF_INC, ILTR_MAX_FIELDLLENGTH+1 );
    if (rc != SUCCESS)
        LOG_ERR_AND_EXIT(rc, TIF_ERR_MEM);

    // Init the date and time formats
    IL_InitStdTimeFormat ( );
    IL_InitStdDateFormat ( );

    // Sanitize ILTR Date Range if invalid (only relevant for Appts and Todos)
    if (ILTR_lDateRangeStart == -1)
        ILTR_lDateRangeStart = 0;

    if (ILTR_lDateRangeEnd == -1)
        ILTR_lDateRangeEnd = 0;

    if (bReInit)
        rc = SecondInit (tr, pstTIF);
    else
        rc = FirstInit (tr, pstTIF, FieldCount);

    // Register Reconciliation Callback Function (separately for ILX32 & ILX16)
    ILRegisterReconCB(tr, TIFReconciliationCallback);

    if (rc == SUCCESS)
        return SUCCESS;

Exit:

    TIFFreeBuffers (tr, pstTIF);

    if (bLogFileOpened)
        ILLOG_end(&TIFLOG);

    IL_FREE_AND_ZERO (hstTIF, *pstTIF);

    return rc;
} //---- TIFInit

/*-----
 * TIFLoadKStruct -- load misc params from Record TWO of workfile
 * Called from:      SecondInit and ILTIFReopenFile
 *                  and TIFSyncCheckOldParameters and ReadOldKStruct
 *
 * returns SUCCESS
 *      or TIF_ERR_BAD_HISTORY_FILE
 *      or TIF_ERR_WRONG_FILE_FORMAT_VERSION
 *      or TIF_ERR_KSTRUCT_SIZE_MISMATCH
 *-----*/
int TIFLoadKStruct (ILDFX_PHNDL phFile, TIF_KSTRUCT *pK)
{
    int rc;
    INT32 len, readlen, ver;

    rc = ILDFX_GetRecordLength (phFile, TIF_RECORD_TWO, &len);
    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_BAD_HISTORY_FILE);

    if (len < sizeof(TIF_KSTRUCT) - 20)
        return ILERROR_L (len, TIF_ERR_BAD_HISTORY_FILE);

    readlen = min (len, sizeof(TIF_KSTRUCT));

    rc = ILDFX_GetRecord ( phFile,
                           TIF_RECORD_TWO,
                           NULL,
                           pK,
                           readlen,
                           NULL ); //-- no ExData for this record
    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_BAD_HISTORY_FILE);
}

```



```

//---- verify that file format version# is current
ver = (INT32) (pK->FileFormatVersion);
if (ver != TIF_CURRENT_FILE_FORMAT_VERSION)
    return ILERROR_L (ver, TIF_ERR_WRONG_FILE_FORMAT_VERSION);

if (len != sizeof(TIF_KSTRUCT))
    return ILERROR_L(len, TIF_ERR_KSTRUCT_SIZE_MISMATCH);

return SUCCESS;
} //---- TIFLoadKStruct

/*-----
 * Name:      WrapUpSetUp
 * Context:   called by TIFStartNextPhase
 * Purpose:   complete setup.  store FieldList in TIF File, etc.
 *-----*/
static int WrapUpSetUp(ILTR_PTRANS� tr, PSTTIF_TYPE pstTIF)
{
    int rc;

    //---- calculate where we'll start storing data in
    //---- record value structs, relative to BYTE ZERO of the record struct
    TIF_FirstOffset(pstTIF) = sizeof(TIFREC_FIXED_PART)
        + (TIF_FieldCount(pstTIF) * sizeof(TIF_FIELD_VALUE));

    TIF_FirstOffset2(TIF_pSourceFieldList) = sizeof(TIFREC_FIXED_PART)
        + (TIF_FieldCount2(TIF_pSourceFieldList) * sizeof(TIF_FIELD_VALUE));

    //---- copy ILTR Date Range Limits into TIF KSTRUCT
    TIF_RangeOfSync.lStartDate = ILTR_lDateRangeStart;
    TIF_RangeOfSync.lEndDate   = ILTR_lDateRangeEnd;

    /*-----
     * If synchronizing, NOT FROM SCRATCH, make sure that's OK.  If not,
     * we may bail out, or may switch to doing a SYNC FROM SCRATCH.
     *-----*/
    if (ILTR_nSynchronize == ILXTR_SYNC_STANDARD)
    {
        char szPath[MAX_PATH];
        IL_MAKEPATH ( szPath, NULL, ILTIF_szHistoryDir, ILTIF_szHistoryName,
                     ILSYNC_EXT_ISH );
        ILDFX_HNDL hHistory; // Handle to open history file

        rc = ILDFX_OpenFile (szPath, &hHistory, ILDFX_MODE_READ);
        if (rc != SUCCESS)
            return rc;

        rc = TIFSyncCheckOldParameters (pstTIF, &hHistory);

        int rc2;
        rc2 = ILDFX_CloseFile (&hHistory, ILDFX_DONT_UPDATE); // rc2 is ignored

        if (rc != SUCCESS)
        {
            TIFlogszul("TIFSyncCheckOldParameters rc=%ld", (UINT32) rc);

            /*-----
             * Put up OK-or-CANCEL dialog to let user choose between cancelling
             * this synchronization run or deleting the current history file
             * and re-synchronizing from scratch.  (see ILSYNC2.CPP)
             *-----*/
            rc = ILTIFSyncShouldWeZapIt(tr);
            if (rc == SUCCESS)
            {
                //--- user said OK; zap history and re-sync
                //--- the 'ZapIt' function has set
                //--- ILTR_nSynchronize == ILXTR_SYNC_FROM_SCRATCH
                ; //...continue...
            }
            else
            {
                //--- user said CANCEL, or an abnormal error has occurred...
                return rc;
            }
        }
    }
}

```

```

/*-----
 * For SYNC FROM SCRATCH, keep Date Range stuff as simple as possible.
 * Avoid triggering any logic that worries about differences between
 * past and present Date Ranges or past & present Dates of Sync.
 *-----*/
if (ILTR_nSynchronize == ILXTR_SYNC_FROM_SCRATCH)
{
    TIF_RangeOfPreviousSync = TIF_RangeOfSync;
    TIF_lDateOfPreviousSync = ILTR_nDate;
}

rc = IdentifyDistinguishedFields(pstTIF);
if (rc != SUCCESS)
    return rc;

rc = IdentifyDefaultFields(tr, pstTIF);
if (rc != SUCCESS)
    return rc;

if (TIF_bCheckForOverlap)
{
    if ( (TIF_StartDateFieldNum == TIF_NOTSET)
        || (TIF_StartTimeFieldNum == TIF_NOTSET)
        || (TIF_EndDateFieldNum == TIF_NOTSET)
        || (TIF_EndTimeFieldNum == TIF_NOTSET) )
    {
        /*-----
         * to check for appointment overlap, we require that
         * Start Date&Time Fields and EndTime Fields be defined. We do
         * not require EndDate field.
         * But rather than returning an error code, we'll just turn off
         * overlap detection.
         *-----*/
        {
            TIFlogsz("Shutting off overlap detection for lack of DTTMs");
            TIF_bCheckForOverlap = FALSE;
            // do not return TIF_ERR_MISSING_OVERLAP_FIELD;
        }
    }
    else
    {
        if (TIF_RelatedFieldNum(pstTIF, TIF_StartDateFieldNum) != TIF_NOTSET)
            /*---- overlap detector can't handle relative start date
            return TIF_ERR_RELATIVE_START_DATE;

        if (TIF_RelatedFieldNum(pstTIF, TIF_StartTimeFieldNum) != TIF_NOTSET)
            /*---- overlap detector can't handle relative start time
            return TIF_ERR_RELATIVE_START_TIME;
    }
}

if (ILTR_nSynchronize == ILXTR_SYNC_STANDARD)
    // don't waste effort on current workfile; it'll soon be overwritten
    return SUCCESS;
else
{
    /*-----
     * Supply EXDATA values for new records -- these values are designed
     * to be eye-catchers and bug-catchers, with no other real function.
     *-----*/
    INT32 exdata[TIF_EXDATA_PER_RECORD];
    int i;

    TIF_FileFormatVersion = TIF_CURRENT_FILE_FORMAT_VERSION;

    // Write out the Target Field List as Record ZERO of Work File
    INT32 lRecSize = sizeof(TIF_FIELDLIST) +
        (sizeof(TIF_FIELD_DESC) * TIF_FieldCount(pstTIF));

    for (i=0; i < TIF_EXDATA_PER_RECORD; i++)
        exdata[i] = 0x7FFFFFF00 + (16 * i) + TIF_RECORD_ZERO;    // eye-catchers

    rc = ILDFX_AddRecord ( TIF_hFile,
        .
        TIF_RECORD_ZERO,
        TIF_pFieldList,
        lRecSize,

```

```

        exdata );

    if (rc == SUCCESS)
    {
        // Write out the Source Field List as Record ONE of Work File
        lRecSize = sizeof(TIF_FIELDLIST) +
            (sizeof(TIF_FIELD_DESC) * TIF_FieldCount2(TIF_pSourceFieldList));

        for (i=0; i < TIF_EXDATA_PER_RECORD; i++)
            exdata[i] = 0x7FFFFFF00 + (16 * i) + TIF_RECORD_ONE;    // eye-catchers

        rc = ILDFX_AddRecord ( TIF_hFile,
                               TIF_RECORD_ONE,
                               TIF_pSourceFieldList,
                               lRecSize,
                               exdata );
    }

    if (rc == SUCCESS)
    {
        // Write out the updated TIF_KSTRUCT as Record TWO of Work File
        lRecSize = sizeof(TIF_KSTRUCT);
        rc = ILDFX_UpdateRecord ( TIF_hFile,
                                   TIF_RECORD_TWO,
                                   &((*pstTIF)->K),
                                   lRecSize,
                                   NULL );    //-- no ExData for this record
    }

    if (rc == SUCCESS)
        rc = ILDFX_GetRecordCount (TIF_hFile, &TIF_TotalRecordCount);

    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_INIT_FAILURE);

    if (TIF_TotalRecordCount != TIF_FIRST_ITEMNO)
        return ILERROR_L (TIF_TotalRecordCount, TIF_ERR_INIT_FAILURE);

    TIFlogszulul (
        "First %ld records in TIF file are CONTROL records; first data item will be %ld",
        TIF_FIRST_ITEMNO, TIF_FIRST_ITEMNO );

    return SUCCESS;
}

} //----- WrapUpSetUp

/*-----
 * Name:      PhaseName  -- for debuglogging only
 *-----*/
static IL_PSTR PhaseName (int phasenum)
{
    switch (phasenum)
    {
        case TIF_PHASE_PREVIOUS:      return "previous";
        case TIF_PHASE_NEXT:          return "next";
        case TIF_PHASE_SETTING_THINGS_UP:    return "Setup";
        case TIF_PHASE_DONE_SETTING_THINGS_UP:    return "Done Setup";
        case TIF_PHASE_LOADING_PREVIOUS_RECORDS:    return "Load Previous History";
        case TIF_PHASE_LOADING_TARGET_RECORDS:    return "Load From Target";
        case TIF_PHASE_LOADING_SOURCE_RECORDS:    return "Load From Source";
        case TIF_PHASE_SANITIZING_SOURCE_RECORDS:    return "Sanitizing Source Records";
        case TIF_PHASE_CHOOSING_RECORDS:    return "Chooser";
        case TIF_PHASE_CONFLICT_RESOLUTION:    return "Conflict Analysis & Resolution";
        case TIF_PHASE_UNLOADING_TO_TARGET:    return "Unload To Target";
        case TIF_PHASE_UNLOADING_TO_SOURCE:    return "Unload To Source";
        case TIF_PHASE_UNLOADING_TO_HISTORY:    return "Unload To History";
        case TIF_PHASE_UNLOADING_FOR_EXPORT:    return "Unload For Export";
        default:
        {
            static char szBuf[20];
            IL_SPRINTF (szBuf, "?? %ld", phasenum);
            return szBuf;
        }
    }
}

```

```

    }

} //---- PhaseName

/*-----
 * Name: SetLoadingRange - set ILTR Date Range for LOADING from System
 *
 * Called from TIFStartNextPhase
 *-----*/
static void SetLoadingRange ( ILTR_PTRANSL tr,
                             PSTTIF_TYPE pstTIF,
                             INT16 phase )
{
    if (ILTR_nAttribs & ILTB_ATT_TOTAL_REBUILD)
    {
        //---- For TOTAL REBUILD systems load ALL records
        ILTR_lDateRangeStart = 0;
        ILTR_lDateRangeEnd   = 0;
    }
    else if ( ( phase == TIF_PHASE_LOADING_TARGET_RECORDS
                && TIF_bTargetIsFastSyncAble )
              || ( phase == TIF_PHASE_LOADING_SOURCE_RECORDS
                && TIF_bSourceIsFastSyncAble ) )
    {
        //---- For a FastSync-capable system load ALL records
        ILTR_lDateRangeStart = 0;
        ILTR_lDateRangeEnd   = 0;
    }
    else
    {
        /*-----
         * For incrementally updateable systems
         * make Loading Range SPAN both Previous & Current Sync Ranges. Normally
         * this gives us the UNION of the 2 ranges, but if Previous & Current
         * Ranges are DISJOINT, we also load from the "gap" in between.
         *-----*/
        ILTR_lDateRangeStart = min ( TIF_RangeOfSync.lStartDate,
                                    TIF_RangeOfPreviousSync.lStartDate );

        ILTR_lDateRangeEnd = max ( TIF_RangeOfSync.lEndDate,
                                   TIF_RangeOfPreviousSync.lEndDate );
    }

    if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 55))
    {
        char sz1[30], sz2[30];

        TIFlogsz3 ( "Loading from DateRange %s -- %s",
                    IL_CodeDateToStdDisplay(ILTR_lDateRangeStart, sz1),
                    IL_CodeDateToStdDisplay(ILTR_lDateRangeEnd, sz2) );
    }
} //---- SetLoadingRange

/*-----
 * Name: ComputeSourceOutcomeCounts
 *
 * Called from TIFStartNextPhase to gather input for OKToProceed.
 *-----*/
static int ComputeSourceOutcomeCounts
( PSTTIF_TYPE pstTIF,
  TIF_OUTCOME_COUNTS *pSourceOutcomeCounts )
{
    ILTR_PTRANSL tr = TIF_tr;

    //---- for SmartMerge simply set all the counts to zero
    if (ILTR_nSynchronize == ILXTR_SYNC_NO)
    {
        IL_MEMSET (pSourceOutcomeCounts, 0, sizeof(TIF_OUTCOME_COUNTS));
        return SUCCESS;
    }

    //---- For Synchronization we have some real work to do

```

```

//---- Set Date Range to ALL
ILTR_lDateRangeStart = 0;
ILTR_lDateRangeEnd   = 0;

//---- do a temporary phase swap for the sake of counting
INT16 savedPhase; savedPhase = TIF_phase;
TIF_phase = TIF_PHASE_UNLOADING_TO_SOURCE;
TIF_CurrentRecordNumber = TIF_POSITION_ABOVE_TOP;

//---- count outcomes and copy counts into caller's structure
int rc;
rc = TIFComputePertinentRecordCount(pstTIF);
if (rc == SUCCESS)
    IL_MEMCPY ( pSourceOutcomeCounts, &TIF_OutcomeCounts,
                sizeof(TIF_OUTCOME_COUNTS) );

//---- restore phase to what it was
TIF_phase = savedPhase;

return rc;
} //---- ComputeSourceOutcomeCounts

/*-----
 * Name:  TIFStartNextPhase
 *       Signals beginning of a new phase and indicates end of previous phase
 * Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
 *
 * NOTE: this function may be called via two different paths, with some
 *       possible overlap, hence redundant calls.  Users may call
 *       ILTIFStartNextPhase directly, and various other calls, such as
 *       ILTIFEndLoad, result in calls to TIFStartNextPhase.
 *
 *       Redundant calls to this function are NO-OPs!!
 *-----*/
int TIFStartNextPhase (ILTR_PTRANSL tr, INT16 phase)
{
    TIF_OUTCOME_COUNTS SourceOutcomeCounts;
    PSTTIF_TYPE pstTIF = &ILTR_pILTIF->pstTIF;
    ILTR_TifPhase = phase;

    /*-----
     * Postpone phase change if workfile isn't open.
     * Phase change will occur when ILTIFReopenFile is called.
     *-----*/
    if (TIF_bWorkFileIsOpen == FALSE)
        return SUCCESS;

    if (phase == TIF_phase)
        return SUCCESS; // redundant call -- NO-OP.

    if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 35))
    {
        TIFlogsz3 ( "\r\n ----- Ending %s phase; starting %s phase\r\n",
                    PhaseName(TIF_phase), PhaseName(phase) );
    }

    int rc;

    /*-----
     * First do any special processing required before EXITING CURRENT PHASE
     *-----*/
    switch (TIF_phase)
    {
        case TIF_PHASE_SETTING_THINGS_UP:
            //----- finish up the setting-up phase...
            rc = WrapUpSetUp(tr, pstTIF);
            if (rc != SUCCESS)
                return rc;
            break;

        case TIF_PHASE_PREVIOUS:
        case TIF_PHASE_DONE_SETTING_THINGS_UP:
    }

```

```

    case TIF_PHASE_LOADING_PREVIOUS_RECORDS:
        break;

    case TIF_PHASE_LOADING_SOURCE_RECORDS:

        /*-----
        * The load-from-source phase that is now ending may or may not
        * have been a "Fast Sync Load". If it was, we normally defer
        * digesting the fast-sync-loaded records until we're done
        * sanitizing them. But if we plan to SKIP the sanitizing step
        * then we need to digest them now.
        *
        * NOTE: deferring digestion until AFTER sanitization may lead
        * to problems. I hope not. But it makes the assumption that
        * all P-items are fully sanitized as far as the Target xlator
        * is concerned. This is because one of the things the
        * "digestion" process does is to supply "missing" Source Items
        * by "cloning" P-Items. So if the case ever arises that a P-item
        * is NOT already fully sanitized, we've got trouble.
        *-----*/
        if ((ILTR_Flags & ILTR_FLAGS_SKIP_SANITIZING_STEP) == 0)
            /*---- Not skipping sanitizing step, so defer digestion,,,
            break;
        else
            ; // Fall through into next case to digest the FastLoad now!!

    case TIF_PHASE_LOADING_TARGET_RECORDS:
    case TIF_PHASE_SANITIZING_SOURCE_RECORDS:

        if (TIF_bFastSyncLoad)
        {
            /*-----
            * The "Fast Sync" flag is ON during Fast Sync Load, but must
            * be OFF while we "Digest" the loaded records. That enables
            * the "Digestion" process to make "normal" PutRecord calls.
            *-----*/
            TIF_bFastSyncLoad = FALSE;
            rc = TIFSyncDigestFastLoad (tr);
            if (rc != SUCCESS)
                return rc;
        }
        break;

    case TIF_PHASE_NEXT:
    case TIF_PHASE_CHOOSING_RECORDS:
    case TIF_PHASE_CONFLICT_RESOLUTION:
    case TIF_PHASE_UNLOADING_TO_TARGET:
    case TIF_PHASE_UNLOADING_TO_SOURCE:
    case TIF_PHASE_UNLOADING_TO_HISTORY:
    case TIF_PHASE_UNLOADING_FOR_EXPORT:
        break;

    default: return TIF_ERR_BAD_CURRENT_PHASE;
}

/*---- For every phase assume FastSync is NOT used, unless changed below...
TIF_bFastSyncLoad = FALSE;
TIF_bFastSyncUnload = FALSE;

/*---- We don't skip "LeaveAlone" items unless decided otherwise below...
TIF_bSkipLeaveAlones = FALSE;

/*-----
* Save Fanout Maxima that apply to the phase we're now exiting,
* if required and not already done by 'ILTIFCloseFileTemporarily'.
*-----*/
if (TIF_bPleaseSaveFanoutMaxima)
{
    TIF_bPleaseSaveFanoutMaxima = FALSE;
    rc = TIFSaveFanoutMaxima (tr);
    if (rc != SUCCESS)
        return rc;
}

```

```

/*-----
 * Now do any special processing required at outset of NEW PHASE
 *-----*/
switch (phase)
{
    case TIF_PHASE_DONE_SETTING_THINGS_UP:
        break;

    case TIF_PHASE_LOADING_PREVIOUS_RECORDS:

        TIF_origin = TIF_FROM_PREVIOUS;
        TIF_phase = phase;

        //---- for this phase set Date Range to ALL
        ILTR_lDateRangeStart = 0;
        ILTR_lDateRangeEnd   = 0;

        rc = TIFSyncReadHistoryFile(tr);
        return rc;

    case TIF_PHASE_LOADING_TARGET_RECORDS:

        TIF_origin = TIF_FROM_TARGET;

        if (ILTR_nSynchronize)
        {
            /*-----
             * Set or Clear the FastSync flag for this phase.  Flag is only set
             * when we're doing "standard" sync (not sync from scratch) and
             * _Delta field is defined in field list.
             * Note that what we set here is overruled if xlator subsequently
             * calls ILTIFFeatureSet to DISABLE FastSync.
             *-----*/
            TIF_bFastSyncLoad = ( (ILTR_nSynchronize == ILXTR_SYNC_STANDARD)
                                && TIF_bTargetIsFastSyncAble );

            //---- Make copy of 'FastSyncLoad' flag in K-struct
            TIF_bFastSyncTargetLoad = TIF_bFastSyncLoad;

            //---- Set Date-related params for Appts & ToDos only
            if (ILTR_nFunction == ILTR_APPT || ILTR_nFunction == ILTR_TODO)
            {
                //---- set Date Range for LOADING from TARGET System
                SetLoadingRange (tr, pstTIF, phase);
                //---- signal that fanout limits should be saved at end of phase
                TIF_bPleaseSaveFanoutMaxima = TRUE;
            }

            /*-----
             * Update Record TWO of Work File, to permanently record the
             * FastSync option used for loading from Target.
             *-----*/
            rc = ILDFX_UpdateRecord ( TIF_hFile,
                                    TIF_RECORD_TWO,
                                    &((*pstTIF)->K),
                                    sizeof(TIF_KSTRUCT),
                                    NULL ); //-- no ExData for this record

            if (rc != ILDFX_OK)
                return rc;
        }
        break;

    case TIF_PHASE_LOADING_SOURCE_RECORDS:

        TIF_origin = TIF_FROM_SOURCE;

        if (ILTR_nSynchronize)
        {
            /*-----
             * Set or Clear the FastSync flag for this phase.  Flag is only set
             * when we're doing "standard" sync (not sync from scratch) and
             * _Delta field is defined in field list.
             * Note that what we set here is overruled if xlator subsequently
             * calls ILTIFFeatureSet to DISABLE FastSync.
             *-----*/

```

```

/*-----*/
TIF_bFastSyncLoad = ( (ILTR_nSynchronize == ILXTR_SYNC_STANDARD)
                      && TIF_bSourceIsFastSyncAble );

//---- Make copy of 'FastSyncLoad' flag in K-struct
TIF_bFastSyncSourceLoad = TIF_bFastSyncLoad;

//---- Set Date-related params for Appts & ToDos only
if (ILTR_nFunction == ILTR_APPT || ILTR_nFunction == ILTR_TODO)
{
    //---- set Date Range for LOADING from SOURCE System
    SetLoadingRange (tr, pstTIF, phase);

    //---- also record SourceLoad Range in KStruct so that it may be
    //---- known to the 'Sanitizing Source Records' phase.
    TIF_RangeOfSourceLoad.lStartDate = ILTR_lDateRangeStart;
    TIF_RangeOfSourceLoad.lEndDate  = ILTR_lDateRangeEnd;

    //---- signal that fanout limits should be saved at end of phase
    TIF_bPleaseSaveFanoutMaxima = TRUE;
}

/*-----*/
* Update Record TWO of Work File. This ensures that the FastSync
* Flag and Range of Source Load will be available when Target
* Translator starts Sanitizing Source Records. (In case TIF file
* has to be closed and re-opened between times.)
/*-----*/
rc = ILDFX_UpdateRecord ( TIF_hFile,
                        TIF_RECORD_TWO,
                        &((*pstTIF)->K),
                        sizeof(TIF_KSTRUCT),
                        NULL ); //-- no ExData for this record

if (rc != ILDFX_OK)
    return rc;
}
break;

case TIF_PHASE_CONFLICT_RESOLUTION:

    TIF_origin = TIF_FROM_SOURCE;
    break;

case TIF_PHASE_UNLOADING_TO_TARGET:

    /*-----*/
    * If we may need to call OKToProceed, compute the source unload count
    * before computing the target unload count. The order isn't critical
    * but we try to avoid confusion and effort wasted counting.
    /*-----*/
    if (ILTR_VERSION_IS_AT_LEAST(28) && ILTR_OKTP_Threshold != 0)
    {
        rc = ComputeSourceOutcomeCounts (pstTIF, &SourceOutcomeCounts);
        if (rc != SUCCESS)
            return rc;
    }

    //---- fall through into next case...

case TIF_PHASE_UNLOADING_TO_SOURCE:

    /*-----*/
    * We choose to do Fast Sync Unload based on the unloader's ABILITY
    * to handle Fast Sync. Doesn't matter whether the LOAD phase was
    * done with FastSync feature enabled or not. When a FastSync-capable
    * translator does a SlowSync LOAD (e.g. to sync from scratch), it
    * must get itself ready for the NEXT sync job to use FastSync. This
    * implies that the consumption of inputs supplied in the SlowSync
    * must be recorded so that we know which if any inputs need to be
    * re-supplied in a subsequent FastSync. So we turn on FastSync-
    * Unload, which causes DeltaAcks to be delivered to the unloader.
    /*-----*/
    if (phase == TIF_PHASE_UNLOADING_TO_TARGET)
        TIF_bFastSyncUnload = (BOOLEAN) TIF_bTargetIsFastSyncAble;
    else

```



```

        TIF_bFastSyncUnload = (BOOLEAN) TIF_bSourceIsFastSyncAble;

/*-----
 * When unloading to a system that does TOTAL REBUILD, we must unload
 * LEAVE_ALONE records as well as ADDS, CHANGES, DELETES, etc. But
 * if system is NOT a TOTAL REBUILD system, we don't let the unloader
 * see the LEAVE_ALONES, cuz we know they're not needed.
 *-----*/
TIF_bSkipLeaveAlones = ((ILTR_nAttribs & ILTB_ATT_TOTAL_REBUILD)==0);

/*-----
 * Turn on USER-VISIBLE logging... (note that EXPORT logging
 * is done in ILTR EXPORT.C, but IMPORT logging that used to
 * be done in ILTR IMPORT.C is now done by ILTIF.)
 *-----*/
TIF_bCurrentlyLoggingAndCountingRecords = TRUE;
//---- fall through into next case...

case TIF_PHASE_UNLOADING_FOR_EXPORT:
case TIF_PHASE_UNLOADING_TO_HISTORY:
case TIF_PHASE_SANITIZING_SOURCE_RECORDS:

    if (ILTR_nSynchronize)
    {
        //---- for these phases, when synchronizing, set Date Range to ALL
        ILTR_lDateRangeStart = 0;
        ILTR_lDateRangeEnd = 0;
    }

    if (phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
    {
        //---- use same FastSyncLoad flag setting as for source load
        TIF_bFastSyncLoad = (BOOLEAN) TIF_bFastSyncSourceLoad;
        TIF_origin = TIF_FROM_SOURCE;
    }
    //---- fall through into next case...

case TIF_PHASE_CHOOSING_RECORDS:

    TIF_phase = phase;
    TIF_CurrentRecordNumber = TIF_POSITION_ABOVE_TOP;
    rc = TIFComputePertinentRecordCount(pstTIF);
    if (rc != SUCCESS)
        return rc;

/*-----
 * Before unloading anything call OKToProceed callback if it exists.
 *-----*/
    if ( (phase == TIF_PHASE_UNLOADING_TO_TARGET)
        && ILTR_VERSION_IS_AT_LEAST(28) && ILTR_OKTP_Threshold != 0)
        rc = TIF_OKToProceed(tr, &SourceOutcomeCounts, &TIF_OutcomeCounts);

    return rc;                // ILTR_ERR_CANCEL if user says no

case TIF_PHASE_NEXT:        // entering unspecified "next" phase is a NO-OP
    break;

case TIF_PHASE_SETTING_THINGS_UP:
default:                    return TIF_ERR_BAD_NEW_PHASE;
}

TIF_phase = phase;
return SUCCESS;
} //---- TIFStartNewPhase

/*-----
 * Name:      TIFSaveFanoutMaxima
 * Purpose:   Copy the Max Fanning Count limits for current phase into KStruct
 *            secure the updates on disk, in TIF_RECORD_TWO of the workfile.
 *
 * NOTE:      we save these parameters at the END of a phase, rather than
 *            doing it earlier, to allow for the possibility that a
 *            translator will change these parameters when it's DataStore

```

```

*      Open function is called. (That's the only LEGAL opportunity
*      that a translator has to override the defaults; has to be done
*      before any records are loaded.)
*
* Callers: TIFStartNextPhase and ILTIFCloseFileTemporarily
*-----*/
int TIFSaveFanoutMaxima (ILTR_PTRANSL tr)
{
    int rc;
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;
    ILDFX_PHNDL phFile = TIF_hFile;

    if (TIF_phase == TIF_PHASE_LOADING_TARGET_RECORDS)
        TIF_TargetFanoutMaxima = ILTR_FanoutMaxima;

    else if (TIF_phase == TIF_PHASE_LOADING_SOURCE_RECORDS)
        TIF_SourceFanoutMaxima = ILTR_FanoutMaxima;

    else
        return ILERROR ((int) TIF_phase, TIF_ERR_WRONG_PHASE);

    /*-----
    * Update Record TWO of Work File, to permanently record the
    * Fanout Maxima that pertain to the current phase
    *-----*/
    rc = ILDFX_UpdateRecord ( TIF_hFile,
                             TIF_RECORD_TWO,
                             &((*pstTIF)->K),
                             sizeof(TIF_KSTRUCT),
                             NULL ); //-- no ExData for this record

    return rc;
} //---- TIFSaveFanoutMaxima

/*-----
* Name:      TIFTerminate
* Purpose:   Shutdown TIF and ILLOG
*            and if we're NOT doing synchronization, DELETE the TIF File!!
*-----*/
int TIFTerminate (PSTTIF_TYPE pstTIF, ILTR_PTRANSL tr)
{
    int rc = SUCCESS;
    int rc2;

    if (*pstTIF == NULL)
        return TIF_PSTTIF_IS_NULL;

    TIFlogsz("TIFTerminate");

    TIFFreeBuffers (tr, pstTIF);

    //---- determine whether we want to KEEP the TIF file.
    //---- if we're going to keep it, we should update it first.
    BOOLEAN bUpdateAndKeepFile;
    bUpdateAndKeepFile = ILTR_VERSION_IS_AT_LEAST(13)
        && (ILTR_Flags & ILTR_FLAG_KEEFILES);

    if (TIF_bWorkFileIsOpen)
    {
        rc = ILDFX_CloseFile (TIF_hFile, bUpdateAndKeepFile);
        if (rc != ILDFX_OK)
            TIFlogszul("ILDFX_CloseFile failed, rc=%ld", (UINT32) rc);
    }

    if (!bUpdateAndKeepFile)
    {
        //----- delete the TIF file
        IL_FILEINFO info; // struct needed to remove file
        IL_REMOVE (TIF_szWorkFile, info, rc2);
        if (rc2 != SUCCESS)
        {
            TIFlogszul("IL_REMOVE failed, rc=%ld", (UINT32) rc2);
            if (rc == SUCCESS)
                //---- no previous error, so remember this error

```

```

        rc = TIF_ERR_CANT_DELETE_FILE;
    }
}

/*-----
 * Close the Debug Logfile
 *-----*/
if (TIFLOG != NULL)
    ILLOG_end(&TIFLOG);

IL_FREE (TIF_hstTIF, *pstTIF);
*pstTIF = NULL;
//----- don't zero TIF_hstTIF cuz it lives inside **pstTIF

return rc;
} //---- TIFTerminate

/*-----
 * Name:      TIFFreeBuffers
 *-----*/
void TIFFreeBuffers (ILTR_PTRANS� tr, PSTTIF_TYPE pstTIF)
{
    // Free Reusable buffers if allocated
    ILUT_FreeBuffer (&TIF_CurrentRecord);
    ILUT_FreeBuffer (&TIF_OriginalRecord);
    ILUT_FreeBuffer (&TIF_FirstRecord);
    ILUT_FreeBuffer (&TIF_SecondRecord);
    ILUT_FreeBuffer (&TIF_SourceRecord);

    ILUT_FreeBuffer (&TIF_CurrentField);
    ILUT_FreeBuffer (&TIF_OriginalField);
    ILUT_FreeBuffer (&TIF_ExtraField);

    // Free fanning buffer if allocated
    if (TIF_pFanBuf != NULL)
    {
        ILUT_FreeBuffer (TIF_pFanBuf);
        IL_FREE (TIF_pFanBuf->hBufferHeader, TIF_pFanBuf);
    }

    // Free Field Lists if allocated
    if (TIF_pFieldList != NULL)
        IL_FREE (TIF_hFieldList, TIF_pFieldList);

    if (TIF_pSourceFieldList != NULL)
        IL_FREE (TIF_hSourceFieldList, TIF_pSourceFieldList);
} //---- TIFFreeBuffers

/*-----
 * Name:      TIFInitRecord
 * Purpose: Initialize the TIF structure representing an ILDFX record buffer
 *-----*/
int TIFInitRecord ( PSTTIF_TYPE          pstTIF,
                   TIF_FIELDLIST_PTR    pFL,
                   ILUT_PBUFFER          pRecord )
{
    int rc;
    ILTR_PTRANS� tr = TIF_tr;
    TIF_RECORD_VALUE_PTR pRecStruct;
    int fieldcount = TIF_FieldCount2(pFL);
    INT16 fieldnum;

    // Calculate specified length according to number of fields
    INT32 lSize = sizeof(TIF_RECORD_VALUE) +
        (sizeof(TIF_FIELD_VALUE) * fieldcount);

    /*-----
     * Allocate record buffer, if not already allocated.
     * Once allocated, record buffer may grow but it never shrinks.
     *-----*/
    if (pRecord->pBuffer == NULL)

```

```

    {
        rc = ILUT_GetBuffer(pRecord, lSize);
        if (rc != SUCCESS)
            return ILERROR_L (lSize, TIF_ERR_MEM);
    }

    pRecStruct = (TIF_RECORD_VALUE_PTR) (pRecord->pBuffer);
    IL_MEMSET (pRecStruct, 0, (size_t) lSize);

    //---- the MEMSET sets all field lengths to ZERO

    //---- initialize all field offsets to NOTSET
    for (fieldnum = 0; fieldnum < fieldcount; fieldnum++)
        TIF_FieldOffset(pRecStruct, fieldnum) = TIF_NOTSET;

    TIFREC_SIZE (pRecStruct) = lSize;
    TIFREC_COMMAND (pRecStruct) = 0;

    /*-----
    * If running under an App that's new enough to use SST stuff,
    * and SST Tagging mechanism is enabled,
    * initialize record subtype to match source section subtype.
    * NOTE: for ILX_V3 mode this is done by ILTR\export.c\ExportWhile.
    *-----*/
    if ( ILTR_VERSION_IS_AT_LEAST(21)
        && ((ILTR_Flags & ILTR_DISABLE_SST_TAGGING) == 0) )
    {
        //--- Get fieldnum needed to initialize the _subType field
        if (pFL == TIF_pSourceFieldList)
            fieldnum = TIF_SourceSubTypeFieldNum;
        else
            fieldnum = TIF_SubTypeFieldNum;

        //---- Initialize record subtype according to where it is coming from
        char szTemp[10];
        int subtype;

        if ( (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
            && (TIF_phase == TIF_PHASE_LOADING_TARGET_RECORDS) )
            subtype = (int) ILTR_TargetSST;
        else
            subtype = (int) ILTR_SourceSST;

        IL_SPRINTF (szTemp, "%d", subtype);

        rc = TIFRecordAddFieldValue
            ( pstTIF, pFL,
              "", fieldnum, // specify field by number
              szTemp,
              0, // length (ignored for all non-binary fields)
              FALSE, // don't map chars
              pRecord );

        if (rc != SUCCESS)
            return ILERROR(rc, TIF_ERR_ABNORMAL);
    } // end if (ILTR_VERSION_IS_AT_LEAST(21))

    return SUCCESS;
} //---- TIFInitRecord

/*-----
* Name:      TIFHowManyField
* Purpose:   Returns how many fields are currently defined in TIF
* Input:     pstTIF
*            Pointer to global information
*            lNumOfFlds
*            Pointer to number of fields defined
* Return:    SUCCESS - If all went well.
*            TIF_ERR_MEM - If there is no global structure
*            lNumFields filled in
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:

```

```

/*-----*/
int TIFHowManyField ( PSTTIF_TYPE pstTIF,
                     INT32 *lNumOfFlds )
{
    *lNumOfFlds = (INT32) TIF_FieldCount(pstTIF);
    return SUCCESS;
} //---- TIFHowManyField

/*-----
* Name:      TIFGetFieldName
* Purpose: Returns field name given an index
* Input:     pstTIF
*           Pointer to global information
*           lFldNdx
*           Field index
*           szFldName
*           Buffer for name of field
* Return:    SUCCESS          - If all went well.
*           TIF_ERR_MEM       - If there is no global structure
*           szFieldName filled in
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:
*-----*/
int TIFGetFieldName ( PSTTIF_TYPE pstTIF,
                     INT16 fieldnum,
                     IL_PSTR szFldName )
{
    if ( (pstTIF == NULL) || (*pstTIF == NULL) )
        return TIF_ERR_PILTIF_IS_NULL;

    // Copy the field name into the buffer
    IL_STRCPY (szFldName, TIF_FieldName(pstTIF, fieldnum));

    return ( SUCCESS );
} //---- TIFGetFieldName

/*-----
* ReflectSourceAttribs -- only caller is TIFDefineOneField
*
* this function does special stuff for Mapped Target Fields for Sync Only.
*-----*/
static int ReflectSourceAttribs ( ILTR_PTRANSL tr,
                                PSTTIF_TYPE pstTIF,
                                int tifFieldnum,
                                ILTR_NDX iltrFieldnum )
{
    //---- Get FieldNum of Source Field that is mapped to this Target Field
    ILTR_PFLDMAP pMap = &ILTR_pTableInfo->sFieldMap;
    ILTR_FLDPTR pFld = &pMap->pTarget[iltrFieldnum];
    ILTR_NDX nSrcFld = pFld->MapField;
    ILTR_FLDPTR pSrcFld;
    ILTB_ATTRIB attribs = TIF_FieldAttributes(pstTIF, tifFieldnum);

    ILTB_ATTRIB a2add = 0;          // attributes to "reflect across" (i.e. add)
    INT32 ea2add = 0;              // Extra Attributes to add

    //---- carry over the NO-RECONCILE
    //---- and INSENSITIVE and STRIPPED COMPARE attributes
    #define CARRYMASK ( ILTB_ATT_NO_RECONCILE      \
                        | ILTB_ATT_INSENSITIVE     \
                        | ILTB_ATT_COMPARE_STRIPPED )

    /*-----
    * First consider top-level fields that are truly mapped; not pseudo-mapped
    * (this first section doesn't see cases where a combined field is mapped
    * by virtue of some of its sub-items being mapped.)
    *-----*/
    if (nSrcFld != ILTR_UNMAPPED && nSrcFld != ILTR_UNMAPPED_BUT_TAGGED)
    {
        //---- Get Ptr to ILTR descriptor for Source Field
        pSrcFld = &pMap->pSource[nSrcFld];
    }

```

```

//---- copy MaxLength of Source Fld into MaxMappedLength for Target Fld
TIF_FieldMaxMappedLength(pstTIF, tifFieldnum) = pSrcFld->Width;

//---- carry over the NO-RECONCILE and INSENSITIVE attributes
a2add |= pSrcFld->Attribs & CARRYMASK;
}

/*-----
 * Next consider combined fields where the top-level combined field is NOT
 * mapped directly. Need to see if any sub-items are mapped.
 *-----*/
else if (attribs & ILTB_ATT_COMBINED)
{
    int i;
    int UnmappedSubItemCount = 0;
    BOOLEAN bL1Mapped; // variable used to check for '1st line mapped' case

    /*-----
     * The first requirement for '1st line mapped' is that the
     * combined field must be a multi-line field.
     *-----*/
    bL1Mapped = ((attribs & ILTB_ATT_MULTLINE) != 0);

    //---- don't run off the end of the fieldmap
    if (iltrFieldnum+1 > pMap->nTarget)
        return SUCCESS;

    //---- take a look at the 1st sub-item of the combined field
    pFld = &pMap->pTarget[iltrFieldnum+1];

    /*-----
     * Nix '1st line mapped' if first sub-item doesn't have
     * terminator=END-OF-LINE.
     *-----*/
    if (pFld->ItemNo != 2 || pFld->Term != ILX_TERM_EOL)
        bL1Mapped = FALSE;

    //---- Nix '1st line mapped' if first sub-item isn't mapped
    nSrcFld = pFld->MapField;
    if (nSrcFld == ILTR_UNMAPPED || nSrcFld == ILTR_UNMAPPED_BUT_TAGGED)
        bL1Mapped = FALSE;

    //---- else carry over the NO-RECONCILE and INSENSITIVE attributes
    else
    {
        pSrcFld = &pMap->pSource[nSrcFld];
        a2add |= pSrcFld->Attribs & CARRYMASK;
    }

    //---- check mapping of subsequent sub-items
    for (i = iltrFieldnum+2; i < pMap->nTarget; i++)
    {
        pFld = &pMap->pTarget[i];

        //---- stop scanning when we run out of sub-items
        if (pFld->ItemNo == 1)
            break;

        //---- Nix '1st line mapped' if we find another mapped sub-item
        nSrcFld = pFld->MapField;
        if (nSrcFld != ILTR_UNMAPPED && nSrcFld != ILTR_UNMAPPED_BUT_TAGGED)
        {
            bL1Mapped = FALSE;
            //---- carry over the NO-RECONCILE and INSENSITIVE attributes
            pSrcFld = &pMap->pSource[nSrcFld];
            a2add |= pSrcFld->Attribs & CARRYMASK;
        }

        //---- count unmapped sub-items
        UnmappedSubItemCount++;
    }

    //---- if the 1 mapped sub-item is followed by 1 or more unmapped
    //---- sub-items, set bit saying field has only first line mapped
    if (bL1Mapped && UnmappedSubItemCount > 0)

```

```

        ea2add |= TIFEA_FIRST_LINE_MAPPED;
    }

    //---- If any attributes need tweaking, do it now
    if (a2add != 0 || ea2add != 0)
    {
        TIFlog2ints(40, "                +%lx.%08lx", ea2add, a2add);
        TIF_FieldAttributes (pstTIF, tifFieldnum) |= a2add;
        TIF_ExtraAttributes (pstTIF, tifFieldnum) |= ea2add;
    }

    return SUCCESS;
} //---- ReflectSourceAttribs

/*-----
 * Name:      TIFDefineOneField
 * Purpose:   Define a field in the TIF
 * Called from: ILTIF.CPP \ ILTIFDefFieldEx
 *
 * NOTE:      currently we use only 1 INT32 worth of ExtraAttributes, but
 *            if more are needed just modify this function and the TIF_FIELD_DESC
 *            structure definition in TIF.H.
 *-----*/
int TIFDefineOneField ( PSTTIF_TYPE pstTIF,
                       TIF_FIELDLIST_PTR pFL,
                       IL_PSTR szFldName,
                       LONG lFldSize,
                       int nFldType,
                       IL_PSTR szFormat,
                       IL_PSTR szRelFldName,
                       ILTB_ATTRIB FieldAttributes,
                       INT32 *pExtraAttributes,
                       BOOLEAN bPositive,
                       IL_PSTR szDefault,
                       ILTR_NDX iltrFieldnum )
{
    INT32 ExtraAttributes = *pExtraAttributes;
    ILTR_PTRANS� tr = TIF_tr;
    int rc;

    if (szFldName == NULL)
        return TIF_ERR_BAD_FLDNAME;

    //---- Increment the actual number of fields. We fill up the fieldlist
    //---- from bottom to top. Get zero-based index into array of fields.
    int fieldnum;
    fieldnum = TIF_FieldCount2(pFL)++;

    if (TIF_FieldCount2(pFL) > pFL->AllocatedFieldCount)
    {
        if (pFL == TIF_pFieldList)
        {
            rc = EnlargeFieldList(pstTIF, &TIF_pFieldList, &TIF_hFieldList);
            pFL = TIF_pFieldList;          // may be relocated!!!
        }
        else
        {
            rc = EnlargeFieldList ( pstTIF,
                                    &TIF_pSourceFieldList,
                                    &TIF_hSourceFieldList );
            pFL = TIF_pSourceFieldList;    // may be relocated!!
        }
        if (rc != SUCCESS)
            return rc;
    }

    /*-----
 * Caller may say he wants to use a Date Field as a Key Field, but
 * when Synchronizing Appts or Todos we relegate Key Date Fields to
 * a slightly inferior status. Keeping DATEs out of the list of
 * bona fide Key Fields, allows SKGs (Same Keyfield Groups) to gather
 * up Fanned Instances and Recurring Masters.
 *-----*/

```

```

* When it comes down to pairing up non-recurring SKG members as
* INEXACT matches, we will refuse to pair up items with different
* 'KeyDateField' values.
*
* See TIFSYNC.CPP - EngageFirstAvailableInSKG and PickFirstInSKG.
*
* NOTE: it is a mistake to designate more than one Date Field as a Key
* Field. The first MAPPED Key Date Field will be treated by TIF
* as the ONLY Key Date Field.
*-----*/
if ( TIF_nSynchronize
    && (FieldAttributes & ILTB_ATT_KEY_FIELD)
    && (nFldType == ILX_TYPE_DATE)
    && (ILTR_nFunction == ILTR_APPT || ILTR_nFunction == ILTR_TODO) )
{
    FieldAttributes &= ~ILTB_ATT_KEY_FIELD;
    ExtraAttributes |= TIFEA_KEY_DATE_FIELD;
    if (pFL == TIF_pFieldList && (ExtraAttributes & TIFEA_ISNT_MAPPED) == 0)
    {
        if (TIF_KeyDateFieldNum == TIF_NOTSET)
        {
            TIF_KeyDateFieldNum = fieldnum;
            TIFlogszszul ( "Field '%s' recognized as KeyDateField (fieldnum=%ld)",
                          szFldName, fieldnum );
        }
        else
            TIFlogsz3 ( "Keeping KeyDateField '%s', ignoring %s",
                       TIF_FieldName(pstTIF, TIF_KeyDateFieldNum), szFldName);
    }
}

/*-----
* Caller may say he doesn't want the Exclusion List for recurring
* items reconciled, but TIF may decide to override that choice.
* Likewise for RepBasic field.
*-----*/
if ( TIF_bReconcileExclusions
    && IL_STRINGS_EQUAL(szFldName, ILTR_REP_XDATE) )
    FieldAttributes &= ~ILTB_ATT_NO_RECONCILE;

/*-----
* Unless filtering by Section SubType is DISABLED, we treat the
* _subType field as a KEY FIELD.
*-----*/
if (IL_STRINGS_EQUAL(szFldName, ILTR_SUB_TYPE))
{
    FieldAttributes &= ~ILTB_ATT_NO_RECONCILE;
    FieldAttributes |= ILTB_ATT_KEY_FIELD;

    if (pFL == TIF_pSourceFieldList)
        TIF_SourceSubTypeFieldNum = fieldnum;
    else
        TIF_SubTypeFieldNum = fieldnum;
}

if ( TIF_bReconcileRepBasic
    && IL_STRINGS_EQUAL(szFldName, ILTR_REP_BASIC) )
    FieldAttributes &= ~ILTB_ATT_NO_RECONCILE;

IL_STRCPY (TIF_FieldName2(pFL, fieldnum), szFldName);
IL_STRCPY (TIF_RelatedFieldName2(pFL, fieldnum), szRelFldName);

TIF_FieldMaxLength2(pFL, fieldnum) = lFldSize;
TIF_FieldType2(pFL, fieldnum) = nFldType;
TIF_FieldNum2(pFL, fieldnum) = fieldnum;

if (szFormat == NULL)
    IL_MAKE_STRING_NULL(TIF_FieldFormat2(pFL, fieldnum));
else
    IL_STRCPY (TIF_FieldFormat2(pFL, fieldnum), szFormat);

/*-----
* Here we simply copy the default value string supplied by the caller.
* Later IdentifyDefaultFields will interpret the default value, in
* case it identifies another field to get the default value from.

```



```

/*-----*/
if (szDefault == NULL)
    IL_MAKE_STRING_NULL(TIF_FieldDefaultValue2(pFL, fieldnum));
else
    IL_STRCPY(TIF_FieldDefaultValue2(pFL, fieldnum), szDefault);

TIF_FieldAttributes2(pFL, fieldnum) = FieldAttributes;
TIF_ExtraAttributes2(pFL, fieldnum) = ExtraAttributes;
TIF_FieldIsAdded2(pFL, fieldnum) = (BOOL16) bPositive;

/*-----*/
* For Synchronization Only, for Mapped Target Fields, pay attention
* to characteristics of the Source Fields to which the Target Fields
* are mapped.
/*-----*/
if ( ILTR_nSynchronize
    && ILTR_phase == ILTR_PHASE10
    && iltrFieldnum != TIF_NOTSET
    && TIF_FieldIsMapped(pstTIF, fieldnum) )
{
    rc = ReflectSourceAttribs (tr, pstTIF, fieldnum, iltrFieldnum);
    if (rc != SUCCESS) return rc;
}

if (ILTR_phase == ILTR_PHASE05)
{
    /*-----*/
    * We currently have some serious deficiencies in field-handling for
    * SOURCE FIELDS. This should only affect Phase 40 of Synchronization.
    * We don't support ValueRequired fields, and may have some problems with
    * use of Related Fields. So here we shut these things off.
    *
    * When some fields are mapped, but others aren't, complications may
    * arise. But a basic approach to removing these restrictions is this:
    *
    * 1. TIF should be told about ALL source fields (or TIF can just
    *    look at the Field Info on its own)
    *
    * 2. When TIF is told that SourceField F gets default value from
    *    SourceField G, it should do the following:
    *
    *    a. determine whether SourceField G is mapped to a
    *       TargetField (TG).
    *
    *    b. if mapped, we get default value from Target field TG.
    *
    *    c. otherwise get default value from Source Field G (with
    *       USFN decorated name!!). Of course if field G is not
    *       set up in TIF, then default value will come straight
    *       from TIF default, not from any field.
    *
    * NOTE that all of a,b,c will be done by TIFIdentifyDefault-
    * Fields, which is called after ALL ILTIFDefField calls have
    * been made.
    *
    * 3. Note that there is no TIFFillDefaults equivalent for Phase40.
    *    Default values must be looked up on the fly by the code
    *    that implements ILTIFGetField. This is actually quite a
    *    complicated thing to do, since ILTIFGetField calls invoke
    *    Field Mapping in Phase40. A possible RESTRICTED approach
    *    might be this (inside ILTIFGetField implementation):
    *
    *    a. call ILFldGet to invoke field mapping.
    *    b. if result of (a) is a null value, and if the Source Field
    *       we're trying to get has the ValueRequired attribute, then
    *       try to get default value using whatever rules are set,
    *       such as trying to get value from another field or using
    *       a constant or TIF default.
    *
    * 4. Concerning Related Fields, ILTR\ILFldGet support for them
    *    should work fine in Phase40, but any TIF-specific code for
    *    dealing with related fields is disabled. This is probably
    *    no great loss.
    /*-----*/
    if (TIF_FieldAttributes2(pFL, fieldnum) & ILTB_ATT_VAL_REQUIRED)

```

```

    {
        TIFlogszsz ("WARNING: turning off ValueRequired attrib for source field '%s'",
szFldName);
        TIF_FieldAttributes2(pFL, fieldnum) &= ~ILTB_ATT_VAL_REQUIRED;
    }

    if (IL_STRING_ISNT_NULL(TIF_RelatedFieldName2(pFL, fieldnum)))
    {
        //--- don't bother warning anyone; it's not worth mentioning cuz it really has
        //--- no impact. Nothing that TIF does during Phase40 depends on TIF's
understanding
        //--- of Related Fields.
        // TIFlogsz3 ( "WARNING: disabling use, by source field %s, of Related Field %s",
        //          szRelFldName, szFldName );
        IL_MAKE_STRING_NULL(TIF_RelatedFieldName2(pFL, fieldnum));
    }
}

return SUCCESS;
} //---- TIFDefineOneField

/*-----
* Name:      TIFLookupFieldDefinition
* Purpose: Locate the definition of a field that TIF knows about
* Author:   David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int TIFLookupFieldDefinition
( PSTTIF_TYPE pstTIF,
  IL_PSTR szFldName,           // IN: Field name
  TIF_FIELD_DESC_PTR IL_DIST *ppDefinition) // OUT: ptr to Definition
{
    // Loop through the fields until we find the one we are looking for
    int fieldnum = TIF_NOTSET;
    int fieldcount = TIF_FieldCount(pstTIF);
    for (int i = 0; i < fieldcount; i++)
    {
        if (IL_STRINGS_EQUAL(szFldName, TIF_FieldName(pstTIF, i)))
        {
            fieldnum = i; break;
        }
    }

    if (fieldnum == TIF_NOTSET)
        return TIF_ERR_BAD_FLDNAME;
    else
    {
        *ppDefinition = &TIF_FieldDesc(pstTIF, fieldnum);
        return SUCCESS;
    }
} //---- TIFLookupFieldDefinition

/*-----
* Name:      TIFPutFieldByIndex -- put field w/o doing any character mapping
* Author:   David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
int TIFPutFieldByIndex
( PSTTIF_TYPE pstTIF,
  INT16 fieldnum,
  IL_PANY pFldData,
  INT32 len ) // len is only relevant for BINARY fields
{
    int rc = TIFRecordAddFieldValue ( pstTIF,
                                      TIF_pFieldList,
                                      "", // field name not specified...
                                      fieldnum, //...specify field by number
                                      (IL_PSTR) pFldData,
                                      len,
                                      FALSE, // do not do character mapping
                                      &TIF_CurrentRecord );

    return rc;
} //---- TIFPutFieldByIndex

```

```

/*-----
* Name:      UseFactoryDefault
* Purpose:   Fill in a TIF default value based on field type.
*           called from TIFFillDefaults
*
* Author:    Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
* Notes:     This routine is called when a value is required for a field
*           and all other attempts at filling this field with a value
*           has failed such as using a supplied default value, and using
*           an associated field default value.(either of which are supplied
*           in the tables.)
*-----*/
static int UseFactoryDefault (PSTTIF_TYPE pstTIF, INT16 fieldnum)
{
    int rc;
    char szDate[9];
    IL_PSTR lpszDefaultValue;

    int nFldType = TIF_FieldType(pstTIF, fieldnum);

    switch ( nFldType )
    {
        case ILX_TYPE_TEXT:
            lpszDefaultValue = "";
            break;

        case ILX_TYPE_BOOL:
            lpszDefaultValue = TIF_BOOL_DEFAULT;
            break;

        case ILX_TYPE_NUMBER:
            lpszDefaultValue = TIF_NUMBER_DEFAULT;
            break;

        case ILX_TYPE_TIME:
            lpszDefaultValue = TIF_TIME_DEFAULT;
            break;

        case ILX_TYPE_DATE:
        {
            int nMonth;
            int nDay;
            int nYear;

            LONG CurrentDate = IL_GetCurrentDate ( );
            IL_DateDecode ( CurrentDate, &nMonth, &nDay, &nYear );
            IL_DateToAlpha ( nMonth, nDay, nYear, szDate );
            lpszDefaultValue = szDate;
            break;
        }

        default:  return TIF_ERR_BAD_FIELD_TYPE;
    }

    rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList, "", fieldnum,
                                lpszDefaultValue,
                                0, // "length" (ignored)
                                FALSE, // do not do character mapping
                                &TIF_CurrentRecord );

    return rc;
} //---- UseFactoryDefault

/*-----
* Name:      TIFFillDefaults
* Purpose:   Make sure that all fields that are supposed to have a value
*           are filled in with either a user supplied default, or if there
*           is no user supplied default then a TIF default value based on
*           the type
* Input:     pstTIF
*           Pointer to global information
*           tr struct
*           hRecord
*-----*/

```

```

*          Handle to the buffer to place the record in
*          pRecord
*          The record buffer
* Return:  SUCCESS      - If all went well.
* Author:  Ken Dobson, Copyright (c) IntelliLink Corporation, 1994
*
* WARNING:  uses the TIF_CurrentField buffer
*-----*/
int TIFFillDefaults ( PSTTIF_TYPE pstTIF,
                     ILTR_PTRANSL tr,
                     ILUT_PBUFFER pRecord )
{
    int rc = SUCCESS;
    LONG lField;                // Length of field data

#define pRecData ((TIF_RECORD_VALUE_PTR) (pRecord->pBuffer))

    // For each of the fields
    for (int fieldnum = 0; fieldnum < TIF_FieldCount(pstTIF); fieldnum++)
    {
        ILTB_ATTRIB FieldAttributes = TIF_FieldAttributes(pstTIF, fieldnum);
        if ((FieldAttributes & ILTB_ATT_VAL_REQUIRED) == 0)
            //---- we don't supply defaults for fields w/o ValueRequired flag
            continue;

        // Retrieve the data for the current field from the current record
        rc = TIFRetrieveFieldByIndex
            (pstTIF, fieldnum, &lField, &TIF_CurrentField, pRecData);
        if (rc != SUCCESS) break;

        if (IL_STRING_ISNT_NULL ((IL_PSTR) TIF_pCurrentField))
            //---- we don't supply defaults for fields that already have values
            continue;

        int nFromField;
        nFromField = (int) TIF_FieldToGetDefaultValueFrom(pstTIF, fieldnum);
        IL_PSTR szDefault; szDefault = TIF_FieldDefaultValue(pstTIF, fieldnum);

        if ((nFromField == TIF_NOTSET) && IL_STRING_ISNT_NULL(szDefault))
        {
            //--- simply plug in constant default value from table
            rc = TIFRecordAddFieldValue
                ( pstTIF,
                  TIF_pFieldList,
                  "", // field name not specified...
                  fieldnum, //...specify field by number
                  szDefault, 0, // NOTE: length (0) is ignored
                  FALSE, // do not do character mapping
                  pRecord );
            if (rc != SUCCESS) break;
        }
        else
        {
            if (nFromField != TIF_NOTSET)
            {
                //--- get default value from another field
                //--- NOTE that this may still leave us with a NULL value
                rc = TIFRetrieveFieldByIndex
                    (pstTIF, nFromField, &lField, &TIF_CurrentField, pRecData);
                if (rc != SUCCESS) break;
            }

            if (IL_STRING_IS_NULL ((IL_PSTR) TIF_pCurrentField))
            {
                rc = UseFactoryDefault (pstTIF, fieldnum);
                if (rc != SUCCESS) break;
            }
            else
            {
                rc = TIFRecordAddFieldValue
                    ( pstTIF,
                      TIF_pFieldList,
                      "", // field name not specified...
                      fieldnum, //...specify field by number
                      (IL_PSTR) TIF_pCurrentField, lField,

```

```

        FALSE, // do not do character mapping
        pRecord );
    if (rc != SUCCESS) break;
}
} // end for (fieldnum...)

return rc;
} //----- TIFFillDefaults

/*-----
* Name:      TIFRetrieveRecord
* Purpose:   Get specified record from TIF
*
* Pass in record# and ptr,handle pair for a reusable/resizable buffer.
* This routine will update the buffer ptr,handle.
*-----*/
int TIFRetrieveRecord ( PSTTIF_TYPE pstTIF,
                      LONG lRecNum,
                      ILUT_PBUFFER pRecord )
{
    ILDFX_EC ec;
    INT32 lRecordLength;

    //TIFlogszul("TIFRetrieveRecord[%ld]", (UINT32) lRecNum);

    if (lRecNum < TIF_FIRST_ITEMNO)
        return ILERROR_L (lRecNum, TIF_ERR_ABNORMAL);

    ec = ILDFX_GetRecordLength (TIF_hFile, lRecNum, &lRecordLength);
    if (ec != ILDFX_OK)
    {
        char sz[40];
        IL_SPRINTF(sz, "recnum=%ld;ec=%d", lRecNum, ec);
        return ILERROR_S (sz, TIF_ERR_ABNORMAL);
    }

    //----- make sure buffer is big enough; expand if necessary
    ec = ILUT_GetBuffer (pRecord, lRecordLength);
    if (ec != SUCCESS)
        return ILERROR_L (lRecordLength, TIF_ERR_MEM);

    ec = ILDFX_GetRecord (TIF_hFile, lRecNum,
                        NULL, // don't want ExData
                        pRecord->pBuffer,
                        lRecordLength,
                        NULL); // don't want bytecount

    return ec;
} //----- TIFRetrieveRecord

/*-----
* Name:      TIFBuildSPT
* Purpose:   Build itty bitty "cig" and "spt" arrays from CIG members.
*           The resulting arrays are useful for picking out CIG members
*           by ORIGIN.
*-----*/
int TIFBuildSPT ( ILDFX_PHNDL phFile,
                 INT32 Item,
                 INT32 *cig,
                 INT32 *spt )
{
    int i;

    cig[0] = TIF_NOTSET;
    cig[1] = TIF_NOTSET;
    cig[2] = TIF_NOTSET;

    spt[TIF_SPT_S] = TIF_NOTSET;
    spt[TIF_SPT_P] = TIF_NOTSET;
    spt[TIF_SPT_T] = TIF_NOTSET;

```

```

    cig[0] = Item;
    cig[1] = TIFGetNextInCIG(phFile, Item);
    if (cig[1] < 0)
        return (int) cig[1]; // abnormal error

    cig[2] = TIFGetNextInCIG(phFile, cig[1]);
    if (cig[2] < 0)
        return (int) cig[2]; // abnormal error

    //----- Now figure things out for Synchronization
    for (i=0; i < 3; i++)
    {
        switch(TIFX_ORIGIN(phFile, cig[i]))
        {
            case TIF_FROM_PREVIOUS: spt[TIF_SPT_P] = cig[i]; break;
            case TIF_FROM_TARGET:   spt[TIF_SPT_T] = cig[i]; break;
            case TIF_FROM_SOURCE:   spt[TIF_SPT_S] = cig[i]; break;
        }
    }

    return SUCCESS;
} //---- TIFBuildSPT

/*-----
* Name:      TIFSetRecordNumbers
* Purpose:   For a given CIG, figure out which item is "current" and which
*            item is "original".
*
* Called from: TIFGetRecord and TIFSyncGetOutcome
*
* NOTE that Current & Original may be identical!!
*-----*/
int TIFSetRecordNumbers ( PSTTIF_TYPE pstTIF,
                        INT32 Item,
                        INT32 IL_DIST *pCurrent,
                        INT32 IL_DIST *pOriginal )
{
    INT32 cig[3];
    INT32 spt[3];
    int rc = TIFBuildSPT (TIF_hFile, Item, cig, spt);
    if (rc != SUCCESS)
        return rc;

    if (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
    {
        /*-----
        * For both SmartMerge and Synchronization, when we're sanitizing source
        * records, all the CIGs we read from are singletons, because we only
        * read un-analyzed records in that phase. In this phase user reads
        * from ORIGINAL and writes to CURRENT.
        *-----*/
        *pCurrent = TIF_NOTSET; // **xref001**
        *pOriginal = cig[0];
    }

    else if (TIF_nSynchronize == ILXTR_SYNC_NO)
    {
        /*-----
        * For SmartMerge, the choice is simple, cuz at this point in SmartMerge
        * there are only two cig types to consider: either a doubleton
        * consisting of a cig[0]=SourceItem and cig[1]=ObsoletedTargetItem, or
        * a singleton. The singleton may be a SourceItem to add, or a
        * TargetItem to leave alone.
        *-----*/
        *pCurrent = cig[0];
        *pOriginal = cig[1];
    }

    else if (cig[0] == cig[1] && cig[1] == cig[2])
    {
        //---- for a singleton CIG things we don't have much choice
        *pCurrent = cig[0];
        *pOriginal = cig[0];
    }
}

```

```

    }

    else
        rc = TIFTablePickRecordsForSync ( pstTIF, Item,
                                           TIF_phase, spt,
                                           pCurrent, pOriginal );

    return rc;
} //---- TIFSetRecordNumbers

/*-----
* Name:      TIFCopyUnmappedFields
* Purpose:   copy unmapped fields from Original record to Current record.
*           This is done when sanitizing source records.
* NOTE:      In addition to unmapped fields, we also copy the _subType field.
*           This ensures that the _subType field value will not be corrupted
*           by the "sanitizing" process. (If someone comes up with a case
*           where a sanitizer needs to be able to modify the _subType, we
*           will frustrate that person until this is changed...)
*-----*/
int TIFCopyUnmappedFields(PSTTIF_TYPE pstTIF)
{
    int rc = SUCCESS;
    int fldcount = TIF_FieldCount(pstTIF);
    for (int fldnum=0; fldnum < fldcount; fldnum++)
    {
        if ( TIF_FieldIsntMapped(pstTIF, fldnum)
            || (fldnum == TIF_SubTypeFieldNum) )
        {
            INT32 len = TIF_FieldLength(TIF_pOriginalRecord, fldnum);
            if (len > 0)
            {
                IL_PSTR pData = TIF_FieldData(TIF_pOriginalRecord, fldnum);
                rc = TIFRecordAddFieldValue ( pstTIF,
                                             TIF_pFieldList,
                                             "", fldnum,
                                             pData,
                                             len,
                                             FALSE, // don't do character mapping
                                             &TIF_CurrentRecord );

                if (rc != SUCCESS) break;
            }
        }
    }
    return rc;
} //---- TIFCopyUnmappedFields

/*-----
* Name:      TIFGetRecord
* Purpose:   Read "current" record into 'CurrentRecord' buffer,
*           and read the original record into the 'OriginalRecord' buffer.
* NOTE:      the RecordNumber passed in by the caller simply points to the CIG;
*           we call 'SetRecordNumbers' to determine actual Current and
*           Original record numbers.
* NOTE that Current & Original may be identical!!
*-----*/
int TIFGetRecord ( PSTTIF_TYPE pstTIF, INT32 RecordNumber )
{
    ILDFX_PHNDL phFile = TIF_hFile;

    int rc = TIFSetRecordNumbers ( pstTIF, RecordNumber,
                                   &TIF_lCurrentRecNum,
                                   &TIF_lOriginalRecNum );

    if (rc != SUCCESS)
        goto TIF_GET_RECORD_EXIT;

    //---- note: if Current Record belongs to a singleton CIG, then
    //---- CurrentRecordNumber == OriginalRecordNumber at this point

    rc = TIFRetrieveRecord ( pstTIF,

```

```

        TIF_lOriginalRecNum,
        &TIF_OriginalRecord );
    if (rc != SUCCESS)
        goto TIF_GET_RECORD_EXIT;

    if (TIF_lCurrentRecNum == TIF_lOriginalRecNum)
    {
        //--- need 2 copies of same record. Instead of reading twice,
        //--- just copy from one buffer into the other.
        rc = TIFCloneRecord (pstTIF);
    }
    else if (TIF_lCurrentRecNum == TIF_NOTSET)
    {
        //--- call TIFInitRecord when Sanitizing Source Records **xref001**
        rc = TIFInitRecord (pstTIF, TIF_pFieldList, &TIF_CurrentRecord);
    }
    else
    {
        rc = TIFRetrieveRecord ( pstTIF,
                                TIF_lCurrentRecNum,
                                &TIF_CurrentRecord );
    }

TIF_GET_RECORD_EXIT:
//-----

    if (rc != SUCCESS)
        TIFlogszulul ( "TIFGetRecord(%ld) rc=%ld",
                        (UINT32) RecordNumber, (UINT32) rc );
    return rc;
} //---- TIFGetRecord

/*-----
* Name:      TIFRetrieveFieldByIndex
* Purpose:   Get specified field data from specified buffer
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
* Notes:     returns TIF_ERR_BAD_FLDNAME when called with fieldnum==TIF_NOTSET.
*
* NOTE:      returned LENGTH includes the NULL terminator for non-binary fields.
*
* Always make sure a buffer is allocated, and return buffer pointer.
* We have to allow for callers who disregard a nonzero return code
* & de-reference the pointer (3/31/95).
*-----*/
int TIFRetrieveFieldByIndex ( PSTTIF_TYPE pstTIF,
                             INT16 fieldNumber,
                             INT32 IL_DIST *plFieldLength, // OUT
                             ILUT_PBUFFER pField,
                             TIF_RECORD_VALUE_PTR pRecord )
{
    int rc = RetrieveFieldValue ( pstTIF,
                                  TIF_pFieldList,
                                  pRecord,
                                  fieldNumber,
                                  plFieldLength, pField );

    return rc;
} //---- TIFRetrieveFieldByIndex

/*-----
* Name:      RetrieveFieldValue
*
* Lowest level FieldValue Retrieval function.
*
* Called by TIFRetrieveFieldByIndex (usually)
* and by TIFGetField for special case of reading from Source Cache.
*-----*/
static int RetrieveFieldValue ( PSTTIF_TYPE pstTIF,
                                TIF_FIELDLIST_PTR pFL,
                                TIF_RECORD_VALUE_PTR pRecord,

```



```

        INT16 fieldNumber,
        INT32 IL_DIST *plFieldLength, // OUT
        ILUT_PBUFFER pField )
{
    int rc, rc2;
    INT32 RequiredBufferSize;
    BOOLEAN bZeroLengthString;
    INT32 FieldOffset;
    INT32 FieldLength;

    //---- fieldlength=zero until successful retrieval of non-null field
    *plFieldLength = 0;

    if (pRecord == NULL)
        rc = TIF_ERR_CANT_READ_FROM_NULL_RECORD;

    else if (fieldNumber == TIF_NOTSET)
        rc = TIF_ERR_BAD_FLDNAME;

    else
    {
        FieldOffset = TIF_FieldOffset(pRecord, fieldNumber);
        if (FieldOffset == TIF_NOTSET)
            rc = SUCCESS; // zero-length field value is OK

        else if (FieldOffset < TIF_NOTSET)
            rc = ILERROR_L (FieldOffset, TIF_ERR_ABSURD_FIELD_OFFSET);

        else
        {
            FieldLength = TIF_FieldLength(pRecord, fieldNumber);
            if ((FieldLength < 0) || (FieldLength > ILTR_MAX_FIELDLNGTH))
                rc = ILERROR_L (FieldLength, TIF_ERR_ABSURD_FIELD_LENGTH);

            else
            {
                if (FieldOffset + FieldLength > TIFREC_SIZE(pRecord))
                {
                    char sz[80];
                    IL_SPRINTF ( sz, "[%ld + %ld > %ld]",
                                FieldOffset, FieldLength, TIFREC_SIZE(pRecord) );
                    rc = ILERROR_S (sz, TIF_ERR_FIELD_EXTENDS_BEYOND_RECORD_END);
                }
                else
                {
                    *plFieldLength = FieldLength;
                    rc = SUCCESS;
                }
            }
        }
    }
}

//---- NOTE: for text fields 'TIF_FieldLength' includes the null terminator
if (*plFieldLength == 0)
{
    /*-----
    * Even though result is zero length, we set minimum buffer size==1
    * to ensure that we always generate a non-null buffer pointer,
    * even for zero-length binary field values.
    *-----*/
    RequiredBufferSize = 1;
    bZeroLengthString = TRUE;
    if ( (fieldNumber != TIF_NOTSET)
        && (TIF_FieldType2(pFL, fieldNumber) != ILX_TYPE_BINARY) )
    {
        /*-----
        * For a zero-length non-binary data value, we actually
        * return a one-byte value: a null-terminated NULL STRING.
        *-----*/
        *plFieldLength = 1;
    }
}
else
{
    RequiredBufferSize = *plFieldLength;
    bZeroLengthString = FALSE;
}

```

```

    }

    rc2 = ILUT_GetBuffer (pField, RequiredBufferSize);
    if (rc2 != SUCCESS)
        return ILERROR_L (RequiredBufferSize, TIF_ERR_MEM);

    if (bZeroLengthString)
        IL_MAKE_STRING_NULL ((IL_PSTR) (pField->pBuffer));
    else
    {
        IL_PSTR lpszFieldData = TIF_FieldData(pRecord, fieldNumber);
        IL_MEMCPY (pField->pBuffer, lpszFieldData, (UINT) *plFieldLength);
    }

    return rc;
} //----- RetrieveFieldValue

/*-----
* Name:      TIFFieldChanged
* Purpose: Determine whether Specified field changed in CURRENT record
*
* WARNING: uses the TIF_CurrentField and TIF_OriginalField buffers
*-----*/
int TIFFieldChanged ( PSTTIF_TYPE pstTIF, IL_PSTR lpszFieldName )
{
    // Loop through the fields until we find the one we are looking for
    int fieldnum = TIF_NOTSET;
    int fieldcount = TIF_FieldCount(pstTIF);
    for (int i = 0; i < fieldcount; i++)
        if (IL_STRINGS_EQUAL(lpszFieldName, TIF_FieldName(pstTIF, i)))
        {
            fieldnum = i;
            break;
        }

    if (fieldnum == TIF_NOTSET)
        return TIF_ERR_BAD_FLDNAME;

    else if (TIF_FieldIsntMapped(pstTIF, fieldnum))
        //----- we don't record history for unmapped fields, so for unmapped
        //----- fields we always say NO, this field value hasn't changed.
        return FALSE;

    else
    {
        BOOLEAN bChanged;
        INT32 CurrentLength, OriginalLength;

        int rc = GetFieldByIndex ( pstTIF, fieldnum, TIF_CURRENT,
                                   &CurrentLength, &TIF_CurrentField );
        if (rc != SUCCESS)
            return rc;

        rc = GetFieldByIndex ( pstTIF, fieldnum, TIF_ORIGINAL,
                                   &OriginalLength, &TIF_OriginalField );
        if (rc != SUCCESS)
            return rc;

        bChanged = TIFFieldValuesDiffer( pstTIF,
                                           (IL_PSTR) TIF_pCurrentField,
                                           (IL_PSTR) TIF_pOriginalField,
                                           CurrentLength,
                                           OriginalLength,
                                           fieldnum );

        return bChanged;
    }
} //----- TIFFieldChanged

/*-----
* Name:      TIFGetField -- get value of named field
*

```

```

* NOTE:  returned LENGTH includes the NULL terminator for non-binary fields.
*-----*/
int TIFGetField ( PSTTIF_TYPE pstTIF,
                  IL_PSTR lpszFieldName,
                  int nWhich,                // original, current, or auto
                  LONG *plFieldLength,       // OUT: length of field value
                  ILUT_PBUFFER pField )     // ptr to buffer header
{
    TIF_FIELDLIST_PTR pFL;

    /*-----
    * First, decide which field list to use for filename lookup.
    *-----*/
    if (nWhich == TIF_SOURCE_CACHE)
        pFL = TIF_pSourceFieldList;
    else
        pFL = TIF_pFieldList;

    // Loop through the fields until we find the one we are looking for
    int fieldnum = TIF_NOTSET;
    int fieldcount = TIF_FieldCount2(pFL);
    for (int i = 0; i < fieldcount; i++)
    {
        if (IL_STRINGS_EQUAL(lpszFieldName, TIF_FieldName2(pFL, i)))
        { fieldnum = i; break; }
    }

    //---- Keep going even if fieldnum is still NOT SET.
    //---- This does lead to a failure code further down the line
    int rc = GetFieldByIndex (pstTIF, fieldnum, nWhich, plFieldLength, pField);
    return rc;
} //---- TIFGetField

/*-----
* Name:      GetFieldByIndex -- get value of numbered field
*
* NOTE:  returned LENGTH includes the NULL terminator for non-binary fields.
*-----*/
static int GetFieldByIndex
( PSTTIF_TYPE pstTIF,
  int fieldnum,
  int nWhich,                // original, current, or auto
  LONG *plFieldLength,       // OUT: length of field value
  ILUT_PBUFFER pField )     // ptr to buffer header
{
    int rc;
    int nSelect;
    TIF_RECORD_VALUE_PTR pRecord;

    /*-----
    * The 'bMexCurrent' flag is only TRUE when caller asks for a CURRENT
    * field value during a 'MEX ONLY' update operation.
    *-----*/
    BOOLEAN bMexCurrent = FALSE;

    if (nWhich == TIF_SOURCE_CACHE)
    {
        /*-----
        * Special Case:  read from Source Cache Record
        *-----*/
        rc = RetrieveFieldValue ( pstTIF,
                                  TIF_pSourceFieldList,
                                  TIF_pSourceRecord,
                                  fieldnum,
                                  plFieldLength, pField );

        return rc;
    }

    /*-----
    * Mainstream code continues, for reading from anything but the Source Cache
    *
    * Next, adjust the selector that tells use
    */

```

```

    * which record to read the field value from...
    *-----*/
    if (fieldnum == TIF_NOTSET)
        /*--- keep going, even though we know the field name is bad...
        nSelect = TIF_CURRENT;

    else if (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
        /*-----
        * when sanitizing source records we always read from ORIGINAL record,
        * regardless of what 'nWhich' the caller asked for.
        *-----*/
        nSelect = TIF_ORIGINAL;

    else
    {
        rc = SelectRecord (pstTIF, fieldnum, nWhich, &nSelect, &bMexCurrent);
        if (rc != SUCCESS)
            return rc;
    }

    /*---- Set pointer to selected record
    if (nSelect == TIF_ORIGINAL)
        pRecord = TIF_pOriginalRecord;
    else
        pRecord = TIF_pCurrentRecord;

    /*-----
    * Finally, retrieve the field value.
    *
    * even if we know that fieldname is bad, go ahead and call the
    * 'ByIndex' retrieve. We do this so that all handling of null
    * results is centralized in a single place. The 'ByIndex'
    * retrieve function returns TIF_ERR_BAD_FLDNAME when called
    * with fieldnum==TIF_NOTSET.
    *-----*/
    rc = TIFRetrieveFieldByIndex ( pstTIF, fieldnum,
                                plFieldLength, pField, pRecord );

    /*-----
    * When we're doing a special exclusion list merge, we adjust the CURRENT
    * exclusion count (in _repBasic) on the fly, right here.
    *-----*/
    if ( rc == SUCCESS && bMexCurrent && fieldnum == TIF_RepBasicFieldNum
        && *plFieldLength == sizeof(ILTR_REPEAT) )
    {
        ILTR_REPEAT pRepeat = (ILTR_REPEAT) (pField->pBuffer);
        INT32 L = TIF_FieldLength (TIF_pCurrentRecord, TIF_RepExclFieldNum);
        pRepeat->numExDates = (INT16) L / sizeof(long);
    }

    return rc;
} /*---- GetFieldByIndex

/*-----
* SelectRecord -- called by GetFieldByIndex
*-----*/
static int SelectRecord ( PSTTIF_TYPE pstTIF,
                        int fieldnum,
                        int nWhich,
                        int *pnSelect,
                        BOOLEAN *pbMexCurrent )
{
    int nSelect;

    switch (nWhich)
    {
        case TIF_ORIGINAL:
            nSelect = TIF_ORIGINAL;
            break;

        case TIF_AUTO:

```

```

if ( TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET
    || TIF_phase == TIF_PHASE_UNLOADING_TO_SOURCE
    || TIF_phase == TIF_PHASE_UNLOADING_TO_HISTORY )
{
    BOOLEAN FieldIsHidden = ((TIF_FieldAttributes(pstTIF, fieldnum)
                             & ILTB_ATT_HIDDEN_FIELD) != 0);
    BOOLEAN OutcomeIsUpdate = ((TIF_CurrentRecordOutcome
                              & ILTIF_OUTCOME_UPDATE) != 0);

    /*-----
    * The TIF_AUTO rules for getting field values during UNLOAD
    * are as follows:
    *
    * When the current record outcome is UPDATE, all UNMAPPED
    * fields come from the ORIGINAL record.
    *
    * No matter what the current record outcome may be,
    * all UNMAPPED HIDDEN fields come from the ORIGINAL record.
    *
    * All MAPPED fields, and all other UNMAPPED fields,
    * come from the CURRENT record.
    *-----*/
    if ( TIF_FieldIsntMapped(pstTIF, fieldnum)
        && (OutcomeIsUpdate || FieldIsHidden) )
        nSelect = TIF_ORIGINAL;
    else
        nSelect = TIF_CURRENT;
    break;
}

//---- .... else fall through into next case...

case TIF_FANNING_ADJ:
case TIF_CURRENT:

    nSelect = TIF_CURRENT;
    break;

default:

    return TIF_ERR_BAD_GETFIELD_NWHICH;
} //---- switch (nWhich)

/*-----
* Adjust selector for special MergeExclusions ONLY case.
* For this situation we get all fields from the ORIGINAL record
* except for two things: the _repExcl field, whose current value
* is in the current record, and the ExclusionCount value in the
* _repBasic field, which is set according to the current _repExcl.
*-----*/
if ( TIF_phase == TIF_PHASE_UNLOADING_TO_TARGET
    || TIF_phase == TIF_PHASE_UNLOADING_TO_SOURCE
    || TIF_phase == TIF_PHASE_UNLOADING_TO_HISTORY )
{
    INT32 Flags = TIFX_FLAGS (TIF_hFile, TIF_lCurrentRecNum);
    *pbMexCurrent = (nSelect == TIF_CURRENT && (Flags & TIF_MEX_ONLY));
    if (*pbMexCurrent && fieldnum != TIF_RepExclFieldNum)
        nSelect = TIF_ORIGINAL;
}

*pnSelect = nSelect;
return SUCCESS;
} //---- SelectRecord

/*-----
* TIFGetViewField
*
* Get pointer to the value of the view field -- for logging
* return NULL ptr if view field is null
*-----*/
int TIFGetViewField (PSTTIF_TYPE pstTIF, IL_PSTR IL_DIST *ppData)
{
    //--- note: we're getting CURRENT value of view field. If preferred,

```

```

//--- we could just as well get ORIGINAL value instead.

//--- assume that TIF_ViewField is ALWAYS SET. By default it
//--- is Field Number Zero.

int fieldnum = TIF_ViewFieldNum;

if (TIF_FieldLength(TIF_pCurrentRecord, fieldnum) == 0)
{
    if (TIF_FieldLength(TIF_pOriginalRecord, fieldnum) == 0)
        *ppData = NULL;
    else
        *ppData = TIF_FieldData(TIF_pOriginalRecord, fieldnum);
}
else
    *ppData = TIF_FieldData(TIF_pCurrentRecord, fieldnum);

return SUCCESS;
} //---- TIFGetViewField

/*-----
 * Name:    ConsiderSmartMergeItem - called by ConsiderThisItem
 *-----*/
static int ConsiderSmartMergeItem ( PSTTIF_TYPE pstTIF,
                                   INT32 Origin, INT32 Flags )
{
    //---- ignore zombies, garbage, and bystanders
    if (Flags & TIF_IGNORE)
        return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;

    if (TIF_phase == TIF_PHASE_CHOOSING_RECORDS)
    {
        //--- for Chooser we want all Source Records. Never count outcomes here.
        if (Origin == TIF_FROM_SOURCE)
            return SUCCESS;
        else
            return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;
    }

    else if (TIF_phase != TIF_PHASE_UNLOADING_TO_TARGET)
        return TIF_ERR_BAD_SMARTMERGE_PHASE;

    //---- If record is from Target, decide what to do with it
    else if (Origin == TIF_FROM_TARGET)
    {
        /*-----
         * If obsoleted (i.e. bumped off by an incoming UPDATE or REPLACE)
         * then skip it; don't even log it.
         *-----*/
        if (Flags & TIF_OUTCOME_OBSOLETE_ITEM)
        {
            if (TIF_bCountingOutcomes) TIF_DeleteCount++;
            return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;
        }

        //--- if LEAVE_ALONE, we may or may not skip it...
        else if ((Flags & TIF_OUTCOME_MASK) == 0)
        {
            if (TIF_bCountingOutcomes) TIF_LeaveAloneCount++;
            /*-----
             * When unloading to a non-total-rebuild system, skip LEAVE_ALONEs
             *-----*/
            if (TIF_bSkipLeaveAlones)
                return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;
            else
                return SUCCESS;
        }

        /*-----
         * The only remaining Target Record possibility is
         * the obsolete DELETE outcome that was used for Achates sync.
         *-----*/
        else
    }

```

```

    {
        if (TIF_bCountingOutcomes) TIF_DeleteCount++;
        return SUCCESS;
    }
}

//--- Record i$ from SOURCE system. Is it to be ignored?
else if (Flags & (TIF_OUTCOME_OBSOLETE_ITEM | TIF_OUTCOME_IGNORED_MASK))
{
    if (TIF_bCountingOutcomes) TIF_IgnoreCount++;

    //---- give ILTIF a chance to log ignored & obsoleted source records
    return TIF_SKIP_THIS_RECORD;
}

//--- else source record will be ADDED to Target System or will be used
//--- to UPDATE or REPLACE a Record in the Target System.
else
{
    if (TIF_bCountingOutcomes)
    {
        if (Flags & TIF_OUTCOME_ADD_ITEM_TO_TARGET)
            TIF_AddCount++;
        else if (Flags & TIF_OUTCOME_REPLACE_ITEM_IN_TARGET)
            TIF_ReplaceCount++;
        else if (Flags & TIF_OUTCOME_UPDATE_ITEM_IN_TARGET)
            TIF_UpdateCount++;
    }
    return SUCCESS;
}
} //---- ConsiderSmartMergeItem

/*-----
* Name:      ConsiderThisItem
* Purpose:   Determine whether item is worth processing or logging
*           ** result depends on whether we're doing synchronization or
*           smartmerge, what phase we're in, and where the item came from.
*
* Called by: TIFPositionToNextRecord,
*           TIFComputePertinentRecordCount,
*           TIFValidateRecord
*
* Returns:   SUCCESS for items that are truly noteworthy
*           or TIF_SKIP_THIS_RECORD for items that are worth logging
*           or TIF_SKIP_AND_DONT_LOG_THIS_RECORD
*           or an abnormal error codes, if something is awry.
*-----*/
static int ConsiderThisItem(PSTTIF_TYPE pstTIF, INT32 Item)
{
    ILDFX_PHNDL phFile = TIF_hFile;

    //---- validate item; complain if invalid item is encountered
    int rc = TIFGroup_ValidateItem(phFile, Item);
    if (rc != SUCCESS)
        return rc;

    INT32 Flags; Flags = TIFX_FLAGS(phFile, Item);
    INT32 Origin; Origin = Flags & TIF_ORIGIN_MASK;

    //---- when we are unloading for export, ALL other items are noteworthy
    if (TIF_phase == TIF_PHASE_UNLOADING_FOR_EXPORT)
        return SUCCESS;

    //---- when sanitizing source records, use all unanalyzed source records
    if (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
    {
        if ((Origin == TIF_FROM_SOURCE) && (Flags & TIF_IS_UNANALYZED))
            return SUCCESS;
        else
            return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;
    }

    //---- Ignore garbage. (NOTE: same bit used for GARBAGE and UNANALYZED)

```

```

if (Flags & TIF_IS_GARBAGE)
    return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;

if (TIF_nSynchronize == ILXTR_SYNC_NO)
{
    //---- Decide what is noteworthy for SmartMerge
    rc = ConsiderSmartMergeItem (pstTIF, Origin, Flags);
    return rc;
}

/*-----
 * Decide what is noteworthy for Synchronization -- depends on cig type.
 *-----*/
INT32 Outcome;
rc = TIFSyncGetOutcome(pstTIF, Item, TIF_phase, &Outcome);
if (rc != SUCCESS)
    return ILERROR_L (Item, rc);

//---- count outcomes, needed for OKToProceed
if (TIF_bCountingOutcomes && Outcome != TIF_SKIP_AND_DONT_LOG_THIS_RECORD)
{
    if (Outcome == TIF_SKIP_FAIL_RANGE)
        TIF_IgnoreCount++;

    else switch (Outcome & ILTIF_FIRST_TIER_OUTCOME_MASK)
    {
        case ILTIF_OUTCOME_LEAVE_ALONE : TIF_LeaveAloneCount++; break;
        case ILTIF_OUTCOME_ADD         : TIF_AddCount++;      break;
        case ILTIF_OUTCOME_DELETE      : TIF_DeleteCount++;   break;
        case ILTIF_OUTCOME_REPLACE     : TIF_ReplaceCount++;   break;
        case ILTIF_OUTCOME_UPDATE      : TIF_UpdateCount++;    break;
        case ILTIF_OUTCOME_IGNORE      : TIF_IgnoreCount++;    break;

        case ILTIF_OUTCOME_LEAVE_DELETED: // don't count these
        default:                          // impossible!!
            break;
    }
}

/*-----
 * Nobody ever wants to deal with records that they're supposed
 * to Leave Deleted, so we never count them or log them or let anyone
 * see them.
 * Also, skip LEAVE_ALONES when unloading to a non-total-rebuild system.
 *
 * Note that LEAVE_DELETED and LEAVE_ALONE outcomes are NOT ignored
 * when OR'd together with the ILTIF_OUTCOME_DELTA_ACK bit.
 *-----*/
if ( (Outcome == ILTIF_OUTCOME_LEAVE_DELETED)
    || (Outcome == TIF_SKIP_AND_DONT_LOG_THIS_RECORD)
    || (Outcome == ILTIF_OUTCOME_LEAVE_ALONE && TIF_bSkipLeaveAlones) )
    return TIF_SKIP_AND_DONT_LOG_THIS_RECORD;

else if (Outcome == TIF_SKIP_FAIL_RANGE)
    return TIF_SKIP_FAIL_RANGE;

else
    return SUCCESS;
} //---- ConsiderThisItem

/*-----
 * Name:      TIFComputePertinentRecordCount
 * Purpose:   Compute the number of records that pertain to the current
 *            UNLOADING PHASE.
 * Called by: TIFStartNextPhase, for start of any UNLOADING PHASE
 * NOTE:      this function does NOT return the count; just computes it!!
 *-----*/
int TIFComputePertinentRecordCount(PSTTIF_TYPE pstTIF)
{
    INT32 TotalRecordCount = TIF_TotalRecordCount;
    INT32 PertinentRecordCount;
    int rc;

```



```

//**** see TIF.H for commentary on 'TIF_TotalRecordCount' ****

if (TIF_phase == TIF_PHASE_UNLOADING_FOR_EXPORT)
    TIF_PertinentRecordCount = TotalRecordCount - TIF_FIRST_ITEMNO;

else
{
    //---- unloading to TARGET, SOURCE, or HISTORY file.
    PertinentRecordCount = 0;

    //---- reset all Outcome Counts to zero
    IL_MEMSET (&TIF_OutcomeCounts, 0, sizeof(TIF_OutcomeCounts));

    //---- tell everyone to count outcomes
    TIF_bCountingOutcomes = TRUE;

    //-----TIF/ILDFX/IndexScanningLoop
    for (INT32 i=TIF_FIRST_ITEMNO; i < TotalRecordCount; i++)
    {
        rc = ConsiderThisItem(pstTIF, i);
        switch (rc)
        {
            case TIF_SKIP_AND_DONT_LOG_THIS_RECORD:
            case TIF_SKIP_THIS_RECORD:
            case TIF_SKIP_FAIL_RANGE:

                continue; // don't count records that we will skip

            case SUCCESS:

                PertinentRecordCount += 1;
                break;

            default:

                TIF_bCountingOutcomes = FALSE;
                return ILERROR(rc, TIF_ERR_ABNORMAL);
        }
    }

    //---- tell everyone to stop counting outcomes
    TIF_bCountingOutcomes = FALSE;
    TIF_PertinentRecordCount = PertinentRecordCount;
} //---- if (TIF_phase == TIF_PHASE_UNLOADING_FOR_EXPORT) .... else ....

/*-----
* Set Goal Posts for THIS pass and NEXT pass. We set both Goal Posts
* to accommodate users who don't call ILTIFHowManyRecords. If there is
* any record-creating activity in THIS pass then the Goal Post for the
* NEXT pass will be incremented.
*
* NOTE: the Goal Post MUST be set as far out as possible to make sure
* that the unloading process can LOG everything that it needs to log.
*
* Bear in mind that TIF_TotalRecordCount is the count of records in
* the ILDFX file, including the TIF "control" records, whether or not
* the TIF control records have been written out to the ILDFX file.
* For example, assuming that there are 2 control records (record #0
* and record #1) and 3 data records (numbered 2, 3, and 4), then
* TIF_TotalRecordCount=5.
*
* The Goal Posts are set to be the zero-based record number of the
* last record in the file, so that an unloader looks all the up to and
* including the last record, but not beyond, where newly created
* records might be found.
*
* Without goalposts we could have a "cannibalism" problem, where an
* unloader might try to unload records that it created by fanning.
*-----*/
TIF_GoalPost = TIF_GoalPostForNextPass = TIF_TotalRecordCount - 1;

return SUCCESS;

```

```

} //----- TIFComputePertinentRecordCount

/*-----
* Name:      TIFPositionToNextRecord
* Purpose:   position to next record that pertains to the current UNLOADING PHASE.
*            (returns TIF_EOF when we run out of pertinent records)
*            (returns TIF_SKIP_THIS_RECORD for source records that are obsoleted
*            or ignored, so that ILTIF can log them...)
*
* NOTE: When EOF is encountered, we turn off the logging of IGNOREs. All
* TIF-assigned IGNOREs are logged by TIF during the first pass of the
* unload. All other outcomes, including cases where TIF suggests
* ADD but the translator decides to IGNORE, are logged when the
* translator calls ILTIFAcceptOutcome or ILTIFRejectOutcome.
*
* TIF_EOF determination depends on 'TIF_GoalPost' which is set by
* TIFComputePertinentRecordCount and ILTIFHowManyRecords.
*-----*/
int TIFPositionToNextRecord(PSTTIF_TYPE pstTIF)
{
    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 GoalPost = TIF_GoalPost;

    if (TIF_CurrentRecordNumber != TIF_POSITION_BELOW_BOTTOM)
    {
        /*-----
        * scan down until we find something worth processing or at least worth
        * logging, or until we reach GoalPost or end of index.
        *-----*/
        for (INT32 i = TIF_CurrentRecordNumber + 1; i <= GoalPost; i++)
        {
            int rc = ConsiderThisItem(pstTIF, i);

            IL_PSTR szrc;

            switch (rc)
            {
                case SUCCESS:                                szrc = "OK";                break;
                case TIF_SKIP_AND_DONT_LOG_THIS_RECORD:      szrc = "Skip&DontLog"; break;
                case TIF_SKIP_THIS_RECORD:                   szrc = "Skip";           break;
                case TIF_SKIP_FAIL_RANGE:                    szrc = "FailRange";     break;
                default:                                     szrc = "Error";         break;
            }

            if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 65))
            {
                char szBuf[80];

                IL_SPRINTF ( szBuf, "Seek Record #%ld (0x%08lx.%03lx) ==> %s (%d)",
                            i, TIFX_FLAGS(TIF_hFile, i),
                            TIFX_FLAGS2(TIF_hFile, i),
                            szrc, rc );
                TIFlogsz(szBuf);
            }

            if (rc != TIF_SKIP_AND_DONT_LOG_THIS_RECORD)
            {
                // Return when we find a noteworthy record.
                // Reasons for noteworthiness are Abnormal Error or record is
                // worth processing or at least worth logging.
                TIF_CurrentRecordNumber = i;
                return rc;
            }
        }

        //----- couldn't find any more pertinent items...
        TIF_CurrentRecordNumber = TIF_POSITION_BELOW_BOTTOM;
    }

    //--- First Pass of current unload phase completed,
    //--- so turn off logging of IGNOREs
    TIF_bCurrentlyLoggingAndCountingRecords = FALSE;
    return TIF_EOF;
}

```

```

} //---- TIFPositionToNextRecord

/*-----
* Name:      TIFValidateRecord
* Purpose: verify that given Record Number identifies a pertinent record,
*           for the current UNLOADING PHASE.
*-----*/
int TIFValidateRecord(PSTTIF_TYPE pstTIF, INT32 RecordNumber)
{
    if ( (RecordNumber < TIF_FIRST_ITEMNO)
        || (RecordNumber >= TIF_TotalRecordCount) )
        return TIF_ERR_BAD_RECORD_NUMBER;

    if (TIF_phase == TIF_PHASE_UNLOADING_FOR_EXPORT)
        return SUCCESS;

    int rc;
    rc = ConsiderThisItem(pstTIF, RecordNumber);
    switch(rc)
    {
        case TIF_SKIP_AND_DONT_LOG_THIS_RECORD:
        case TIF_SKIP_THIS_RECORD:
        case TIF_SKIP_FAIL_RANGE:

            return TIF_ERR_IMPERTINENT_RECORD;

        default:

            return rc; // SUCCESS or abnormal error
    }
} //---- TIFValidateRecord

/*-----
* Name:      TIFGetOutcome
* Purpose: Determine whether current record is to be Replaced, Updated,
*           Deleted, Added, or Just Left Alone.
*-----*/
int TIFGetOutcome (PSTTIF_TYPE pstTIF, INT32 Item,
                  INT32 IL_DIST *pOutcome)
{
    ILDFX_PHNDL phFile = TIF_hFile;
    int rc = SUCCESS;

    if ( (TIF_phase != TIF_PHASE_UNLOADING_TO_TARGET)
        && (TIF_phase != TIF_PHASE_UNLOADING_TO_SOURCE)
        && (TIF_phase != TIF_PHASE_UNLOADING_TO_HISTORY) )
    {
        /*-----
        * When we're not in an UNLOADING phase, we don't have an outcome
        * to brag about. Just set it to zero and don't log this call.
        *-----*/
        *pOutcome = 0;
        return SUCCESS;
    }
    else if (TIF_nSynchronize)
        /*----- set Synchronization outcome for record
        rc = TIFSyncGetOutcome(pstTIF, Item, TIF_phase, pOutcome);
    else
    {
        /*--- set SmartMerge outcome for record
        INT32 OutcomeBits = TIFX_OUTCOME(phFile, Item);

        if (TIFX_ORIGIN(phFile, Item) == TIF_FROM_TARGET)
        {
            if (OutcomeBits == 0)
                *pOutcome = ILTIF_OUTCOME_LEAVE_ALONE;
            else if (OutcomeBits & TIF_OUTCOME_DELETE_ITEM_FROM_TARGET)
                *pOutcome = ILTIF_OUTCOME_DELETE;
            else if (OutcomeBits & TIF_OUTCOME_OBSOLETED_ITEM)
                *pOutcome = ILTIF_OUTCOME_OBSOLETED;
            else
                rc = TIF_ERR_IMPOSSIBLE_OUTCOME;
        }
    }
}

```

```

    }
    else
    //----- else record must be from PREVIOUS or from SOURCE...
    {
        if (OutcomeBits & TIF_OUTCOME_OBSOLETE_ITEM)
            *pOutcome = ILTIF_OUTCOME_OBSOLETE;
        else if (OutcomeBits & TIF_OUTCOME_ADD_ITEM_TO_TARGET)
            *pOutcome = ILTIF_OUTCOME_ADD;
        else if (OutcomeBits & TIF_OUTCOME_REPLACE_ITEM_IN_TARGET)
            *pOutcome = ILTIF_OUTCOME_REPLACE;
        else if (OutcomeBits & TIF_OUTCOME_UPDATE_ITEM_IN_TARGET)
            *pOutcome = ILTIF_OUTCOME_UPDATE;
        else if (OutcomeBits & TIF_OUTCOME_IGNORED_MASK)
            *pOutcome = ILTIF_OUTCOME_IGNORE;
        else
            rc = TIF_ERR_IMPOSSIBLE_OUTCOME;
    }
}

if (rc != SUCCESS)
    TIFlogszul ("TIFGetOutcome rc=%ld", (UINT32) rc);

return rc;
} //----- TIFGetOutcome

/*-----
* Name:      LogILTRFieldList
*
* Called from FirstInit, which is called from TIFInit
*
* Purpose: for DEBUG use only; validate assumptions about ILTR field lists
*-----*/
static int LogILTRFieldList(ILTR_PTRANSL tr)
{
    ILTR_PFLDMAP pMap;
    ILTR_FLDPTR p;
    char szBuf[150];

    ILTIFlogsz("Target Field Map:");

    if (ILTR_phase == ILTR_PHASE_ILX_V3_MODE)
        pMap = &ILTR_map;
    else
        pMap = &ILTR_pTableInfo->sFieldMap;

    for (int i=0; i < pMap->nTarget; i++)
    {
        p = &pMap->pTarget[i];
        IL_SPRINTF(szBuf, "%d.%d: %s %s (%c, %s, %d)",
                    i, p->ItemNo, p->IntName, p->ExtName,
                    p->Type, p->TypeDesc, p->MapField);
        ILTIFlogsz(szBuf);
    }

    if (ILTR_VERSION_IS_AT_LEAST(14))
    {
        ILTIFlogsz("Extra Special Fields:");
        int SpecialBase = pMap->nTarget;
        int SpecialLimit = pMap->nTarget + ILTR_pTableInfo->nExtraFields;
        for (int i=SpecialBase; i < SpecialLimit; i++)
        {
            p = &pMap->pTarget[i];
            IL_SPRINTF(szBuf, "%d.%d: %s %s (%c, %s, %d)",
                        i, p->ItemNo, p->IntName, p->ExtName,
                        p->Type, p->TypeDesc, p->MapField);
            ILTIFlogsz(szBuf);
        }
    }

    return SUCCESS;
} //----- LogILTRFieldList

```

```

/*-----
* Name:      TIFDUMP.CPP
* Purpose:   for debugging, produced formatted dump of TIF contents
*
* Function (local functions indented):
*
*           tifDumpGetRecordName
*           dump_and_unmark_cig
*           dump_and_unmark_skg
*           TIFDump
*           AddToXlateLog
*           dump2_and_unmark_cig
*           TIFDump2
*           TIFCigName
*
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995-1996
*-----*/

#include "iltr.h"

#define TIFDUMP_MAXNAME 30

typedef struct
{
    INT32 ClashCount;
    INT32 ChgCount;
    INT32 DelCount;
    INT32 AddCount;
    INT32 NoChgCount;
}
TIF_SUMS;

/*-----
* Name:      tifDumpGetRecordName
*-----*/
static int tifDumpGetRecordName (PSTTIF_TYPE pstTIF, INT32 recnum,
                                IL_PSTR szRecordName)
{
    int rc = TIFRetrieveRecord (pstTIF, recnum, &TIF_CurrentRecord);
    if (rc != SUCCESS)
        return rc;

    //---- Retrieve the view field from the current record
    INT32 fieldlen;
    rc = TIFRetrieveFieldByIndex ( pstTIF, TIF_ViewFieldNum,
                                &fieldlen,
                                &TIF_CurrentField,
                                TIF_pCurrentRecord );

    if (rc != SUCCESS) return rc;

    IL_SAFE_STRINGCOPYN ( szRecordName,
                        (IL_PSTR) TIF_pCurrentField,
                        TIFDUMP_MAXNAME );

    return SUCCESS;
} //---- tifDumpGetRecordName

/*-----
* Name:      dump_and_unmark_cig
*-----*/
int dump_and_unmark_cig (PSTTIF_TYPE pstTIF, INT32 groupNumber, INT32 anchor)
{
    ILDEX_PHNDL phFile = TIF_hFile;
    INT32 j=anchor;
    int i;

    for (i=1; i <= TIF_MAX_CIG_SIZE; i++)    // catch infinite loops
    {
        INT32 lFlags;
        INT32 lFlags2;
        INT32 lSourceIDHash;
        INT32 lTargetIDHash;
        INT32 lKeyFieldsHash;
        INT32 lNonKeyFieldsHash;
    }

```

```

    INT32 lDescHash;
    INT32 lNextInFIG;

    TIFX_FLAGS2(phFile, j) &= ~TIF2_MARK_DUMP1;

    lFlags = TIFX_FLAGS(phFile, j);
    lFlags2 = TIFX_FLAGS2(phFile, j);

    INT32 next = TIFGetNextInCIG(phFile, j);
    if (next < 0)
        return (int) next; // next node failed sanity check!!

    if (next == j)
        break; // this is a singular CIG; don't dump it

    lSourceIDHash = TIFX_SOURCEID_HASH(phFile, j);
    lTargetIDHash = TIFX_TARGETID_HASH(phFile, j);
    lKeyFieldsHash = TIFX_KEYFIELDS_HASH(phFile, j);
    lNonKeyFieldsHash = TIFX_NKFIELDS_HASH(phFile, j);
    lDescHash = TIFX_DESC_HASH(phFile, j);
    lNextInFIG = TIFX_NEXT_IN_FIG(phFile, j);

    char szName[TIFDUMP_MAXNAME];
    if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 70))
    {
        int rc = tifDumpGetRecordName (pstTIF, j, szName);
        if (rc != SUCCESS) return rc;
    }
    else
        IL_MAKE_STRING_NULL(szName);

    char szBuf[140];

    if (TIF_nSynchronize)
        IL_SPRINTF(szBuf,
"CIG#%3ld i=%3ld SIH=%8lx TIH=%8lx KFH=%8lx NKH=%8lx NIF=%3ld F=%8lx.%04lx %s",
        groupNumber, j, lSourceIDHash, lTargetIDHash,
        lKeyFieldsHash, lNonKeyFieldsHash, lNextInFIG,
        lFlags, lFlags2, szName);
    else
        IL_SPRINTF(szBuf,
"CIG#%3ld i=%3ld SIH=%8lx TIH=%8lx KFH=%8lx NKH=%8lx DH=%8lx F=%8lx.%03lx %s",
        groupNumber, j, lSourceIDHash, lTargetIDHash,
        lKeyFieldsHash, lNonKeyFieldsHash, lDescHash,
        lFlags, lFlags2, szName);
    TIFlogsz(szBuf);

    j = next;
    if (j == anchor)
        break; // have come full circle.
}

if (i > TIF_MAX_CIG_SIZE) // catch infinite loops
    return TIF_ERR_BROKEN_CIG;

return SUCCESS;
} //---- dump_and_unmark_cig

/*-----
* Name:    dump_and_unmark_skg
*-----*/
int dump_and_unmark_skg (PSTTIF_TYPE pstTIF, INT32 groupNumber, INT32 anchor)
{
    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 j=anchor;
    int i;

    for (i=1; i <= TIF_MAX_SKG_SIZE; i++) // catch infinite loops
    {
        INT32 lFlags;
        INT32 lFlags2;
        INT32 lSourceIDHash;
        INT32 lTargetIDHash;

```

```

    INT32 lKeyFieldsHash;
    INT32 lNonKeyFieldsHash;
    INT32 lDescHash;
    INT32 lNextInFIG;

    TIFX_FLAGS2(phFile, j) &= ~TIF2_MARK_DUMP2;

    lFlags = TIFX_FLAGS(phFile, j);
    lFlags2 = TIFX_FLAGS2(phFile, j);

    lSourceIDHash = TIFX_SOURCEID_HASH(phFile, j);
    lTargetIDHash = TIFX_TARGETID_HASH(phFile, j);
    lKeyFieldsHash = TIFX_KEYFIELDS_HASH(phFile, j);
    lNonKeyFieldsHash = TIFX_NKFIELDS_HASH(phFile, j);
    lDescHash = TIFX_DESC_HASH(phFile, j);
    lNextInFIG = TIFX_NEXT_IN_FIG(phFile, j);

    char szName[TIFDUMP_MAXNAME];
    if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 70))
    {
        int rc = tifDumpGetRecordName (pstTIF, j, szName);
        if (rc != SUCCESS)
            return rc;
    }
    else
        IL_MAKE_STRING_NULL(szName);

    char szBuf[140];

    if (TIF_nSynchronize)
        IL_SPRINTF(szBuf,
            "SKG#%3ld i=%3ld SIH=%8lx TIH=%8lx KFH=%8lx NKH=%8lx NIF=%3ld F=%8lx.%04lx %s",
            groupNumber, j, lSourceIDHash, lTargetIDHash,
            lKeyFieldsHash, lNonKeyFieldsHash, lNextInFIG,
            lFlags, lFlags2, szName);
    else
        IL_SPRINTF(szBuf,
            "SKG#%3ld i=%3ld SIH=%8lx TIH=%8lx KFH=%8lx NKH=%8lx DH=%8lx F=%8lx.%03lx %s",
            groupNumber, j, lSourceIDHash, lTargetIDHash,
            lKeyFieldsHash, lNonKeyFieldsHash, lDescHash,
            lFlags, lFlags2, szName);

    TIFlogsz(szBuf);

    j = TIFGetNextInSKG(phFile, j);
    if (j < 0)
        return (int) j; // next node failed sanity check!!

    if (j == anchor)
        break; // we have come full circle
}

if (i > TIF_MAX_SKG_SIZE) // catch infinite loops
    return TIF_ERR_BROKEN_SKG;

return SUCCESS;
} //---- dump_and_unmark_skg

/*-----
 * Name:      TIFDump
 * For Debugging, create formatted dump of
 * the CURRENTLY OPEN TIF file
 *-----*/
int TIFDump (PSTTIF_TYPE pstTIF)
{
    ILDFX_PHNDL phFile = TIF_hFile;
    ILDFX_EC ec;
    INT32 lCount;

    TIFlogsz("\r\n----- TIF DUMP ----- \r\n");
    TIFlogsz("Corresponding Item Groups: \r\n");

    ec = ILDFX_GetRecordCount (phFile, &lCount);

```

```

TIFlogszulul("ILDFX_GetRecordCount ==> count=%ld, ec=%ld",
             (UINT32) lCount, (UINT32) ec);
if (ec != ILDFX_OK)
    return ec;

INT32 i;
INT32 lFlags;
INT32 lFlags2;

//----- mark ALL records (we unmark as we dump)
for (i=TIF_FIRST_ITEMNO; i < lCount; i++)
    TIFX_FLAGS2(phFile, i) |= (TIF2_MARK_DUMP1 | TIF2_MARK_DUMP2);

//----- now dump all CIGs
INT32 lGroup;
lGroup = 0;
//-----TIF/ILDFX/IndexScanningLoop
for (i=TIF_FIRST_ITEMNO; i < lCount; i++)
{
    lFlags = TIFX_FLAGS(phFile, i);
    lFlags2 = TIFX_FLAGS2(phFile, i);

    //---- don't log zombies or garbage or totally out-of-range stuff here
    if ( (lFlags2 & TIF2_MARK_DUMP1)
        && ((lFlags & TIF_IGNORE) == 0)
        && ((lFlags2 & TIF2_WAS_OUTRANGE_10X) == 0) )
    {
        lGroup += 1;
        ec = dump_and_unmark_cig (pstTIF, lGroup, i);
        if (ec != ILDFX_OK)
            return ec;
    }
}

TIFlogsz("\r\nSame Keyfield Groups:\r\n");

//----- now dump all SKGs
lGroup = 0;
BOOLEAN bSingleton;
bSingleton = FALSE;
//-----TIF/ILDFX/IndexScanningLoop
for (i=TIF_FIRST_ITEMNO; i < lCount; i++)
{
    lFlags2 = TIFX_FLAGS2(phFile, i);
    if (lFlags2 & TIF2_MARK_DUMP2)
    {
        //---- don't log zombies if verbosity level is less than 88
        lFlags = TIFX_FLAGS(phFile, i);
        if ( ((lFlags & TIF_IGNORE) || (lFlags2 & TIF2_WAS_OUTRANGE_10X))
            && !ILLOG_VERBOSE_ENOUGH(TIFLOG, 88))
            continue;

        if (TIFX_NEXT_IN_SKG(phFile, i) == i)
            bSingleton = TRUE;
        else
        {
            if (bSingleton)
                TIFlogsz(" "); // double space between singleton and non-s.
            bSingleton = FALSE;
        }

        lGroup += 1;
        ec = dump_and_unmark_skg (pstTIF, lGroup, i);
        if (ec != ILDFX_OK)
            return ec;
        if (!bSingleton)
            TIFlogsz(" "); // double space after non-singleton
    }
}

return SUCCESS;
} //----- TIFDump

```



```

/*-----
 * Name:      dump2_and_unmark_cig
 *-----*/
static int AddToXlateLog (PSTTIF_TYPE pstTIF, INT32 Item, int nMsgId)
{
    ILTR_PTRANSL tr = TIF_tr;
    char szName[TIFDUMP_MAXNAME];
    char szAnalysis[100];
    char szLogEntry[150];

    //---- get "name" of record
    int rc = tifDumpGetRecordName (pstTIF, Item, szName);
    if (rc != SUCCESS) return rc;

    //---- Truncate multi-line names at end of first line.
    IL_PSTR lpMatch;
    lpMatch = IL_STRCHR (szName, ILTR_EOS_CHAR);
    if (lpMatch != NULL)
        *lpMatch = 0;

    //---- get string that says what we know about this record
    rc = ILSTMakeString (&hXlatorInst, nMsgId, szAnalysis, sizeof (szAnalysis));

    //---- put record name and record analysis together
    IL_SPRINTF (szLogEntry, " %-30.30s - %s", szName, szAnalysis);
    IL_WRITE (ILTR_hLog, szLogEntry, IL_STRLEN(szLogEntry), rc);

    //---- ignore difficulties writing to log file (bad idea?)
    return SUCCESS;
} //---- AddToXlateLog

/*-----
 * Name:      dump2_and_unmark_cig
 *-----*/
static int dump2_and_unmark_cig ( PSTTIF_TYPE pstTIF, INT32 groupNumber,
                                INT32 First, TIF_SUMS& Sums )
{
    int rc = SUCCESS;
    INT32 cigType;
    INT32 mexType = 0;
    INT32 Second;
    INT32 Third;
    INT32 Sanity;
    ILDFX_PHNDL phFile = TIF_hFile;
    int nMsgId;

    TIFX_FLAGS2(phFile, First) &= ~TIF2_MARK_DUMP1;
    cigType = TIFX_CIG_TYPE(phFile, First);

    Second = TIFGetNextInCIG(phFile, First);
    if (Second < 0)
        return (int) Second; // next node failed sanity check!!

    if (Second != First)
    {
        TIFX_FLAGS2(phFile, Second) &= ~TIF2_MARK_DUMP1;
        if (TIFX_CIG_TYPE(phFile, Second) != cigType)
            return ILERROR_L (Second, TIF_ERR_HETEROGENEOUS_CIG);

        Third = TIFGetNextInCIG(phFile, Second);
        if (Third < 0)
            return (int) Third; // next node failed sanity check!!

        if (Third == First)
        {
            //---- need to get ordering of 2-item CIGs into PTS order
            //---- re-ordering is only required when Source data is loaded
            //---- before Target data. Previous data, if any, is ALWAYS loaded first.
            if (TIFX_ORIGIN(phFile, First) == TIF_FROM_SOURCE)
            {
                INT32 tmp = First; // swap Target & Source items
                First = Second;
                Second = tmp;
            }
        }
    }
}

```

```

    }
}
else
{
    TIFX_FLAGS2(phFile, Third) &= ~TIF2_MARK_DUMP1;
    if (TIFX_CIG_TYPE(phFile, Third) != cigType)
        return ILERROR_L (Third, TIF_ERR_HETEROGENEOUS_CIG);
    Sanity = TIFGetNextInCIG(phFile, Third);
    if (Sanity < 0)
        return (int) Sanity; // next node failed sanity check!!

    if (Sanity != First)
        return TIF_ERR_BROKEN_CIG; // failed to circle back to 1st item
}
}

/*-----
 * If CIG type has been altered to Merge Exclusion Lists, log the original
 * CIG type, (NOTE: in TIF.LOG we log both.)
 *-----*/
if (TIFX_FLAGS(phFile, First) & TIF_MEX_CIG)
{
    mexType = cigType;
    cigType = TIFX_FLAGS2(phFile, First) & TIF2_ORIGINAL_CIG_TYPE_MASK;
}

/*-----
 * Now decide how to count and log this CIG. The counting we do here is
 * only for display in the user-visible logfile. We do two different
 * types of logging here: DEBUG-logging and user-visible logging.
 *
 * When conflicts that start out as CIG types 102 and 213 are resolved
 * by the ADD-ACROSS option, the workfile at this point contains fragments
 * of the original CIGs. A "102" breaks down into a "100" and a "001".
 * A "213" breaks down into a "100" and a "010" and a "001".
 *
 * The TIF2_WAS_CIGTYPE_xxx flag bits tell us how we got here, so we can
 * accurately reflect the original reality in the user-visible logfile.
 * For each group of fragments we only log the "100" fragment.
 *-----*/
switch (cigType)
{
    case TIF_CIG_TYPE_010:

        //---- don't user-visibly count or log internally generated 010 CIGs
        if (TIFX_FLAGS2(phFile, First) & TIF2_WAS_CIGTYPE_213)
        {
            TIFloglint ( 25,
                "--- %3ld --- <213:010> Source+Target DELETE", First );
            nMsgId = TIF_NOTSET;
        }
        else
        {
            Sums.DelCount++;
            TIFloglint(25, "--- %3ld --- <010> Source+Target DELETE", First);
            nMsgId = TIF_STR_CIG010;
        }
        break;

    case TIF_CIG_TYPE_001:

        //---- don't user-visibly count or log internally generated 001 CIGs
        if (TIFX_FLAGS2(phFile, First) & TIF2_WAS_CIGTYPE_213)
        {
            TIFloglint (25, "--- --- %3ld <213:001> Target ADD", First);
            nMsgId = TIF_NOTSET;
        }
        else if (TIFX_FLAGS2(phFile, First) & TIF2_WAS_CIGTYPE_102)
        {
            TIFloglint (25, "--- --- %3ld <102:001> Target ADD", First);
            nMsgId = TIF_NOTSET;
        }
        else
        {
            Sums.AddCount++;

```

```

        TIFloglint (25, "--- --- %3ld <001> Target ADD", First);
        nMsgId = TIF_STR_CIG001;
    }
    break;

case TIF_CIG_TYPE_100:
    //---- special counting & logging for internally generated 100 CIGs
    if (TIFX_FLAGS2(phFile, First) & TIF2_WAS_CIGTYPE_213)
    {
        Sums.ClashCount++;
        TIFloglint (25, "%3ld --- --- <213:100> Source ADD", First);
        nMsgId = TIF_STR_CIG213;
    }
    else if (TIFX_FLAGS2(phFile, First) & TIF2_WAS_CIGTYPE_102)
    {
        Sums.ClashCount++;
        TIFloglint (25, "%3ld --- --- <102:100> Source ADD", First);
        nMsgId = TIF_STR_CIG102;
    }
    else
    {
        Sums.AddCount++;
        TIFloglint (25, "%3ld --- --- <100> Source ADD", First);
        nMsgId = TIF_STR_CIG100;
    }
    break;

case TIF_CIG_TYPE_011:
    Sums.DelCount++;
    TIFlog2ints ( 25, "--- %3ld %3ld <011> Source DELETE",
        First, Second );
    nMsgId = TIF_STR_CIG011;
    break;

case TIF_CIG_TYPE_012:
    //--- count this as a DELETE.  Some day that may change!!
    Sums.DelCount++;
    TIFlog2ints ( 25, "--- %3ld %3ld <012> CONFLICT: DELETE vs CHANGE",
        First, Second );
    nMsgId = TIF_STR_CIG012;
    break;

case TIF_CIG_TYPE_110:
    Sums.DelCount++;
    TIFlog2ints ( 25, "%3ld %3ld --- <110> Target DELETE",
        Second, First );
    nMsgId = TIF_STR_CIG110;
    break;

case TIF_CIG_TYPE_210:
    //--- count this as a DELETE.  Some day that may change!!
    Sums.DelCount++;
    TIFlog2ints ( 25, "%3ld %3ld --- <210> CONFLICT: CHANGE vs DELETE",
        Second, First );
    nMsgId = TIF_STR_CIG210;
    break;

case TIF_CIG_TYPE_101:
    Sums.NoChgCount++;
    TIFlog2ints ( 25, "%3ld --- %3ld <101> Source+Target ADD",
        Second, First );
    nMsgId = TIF_STR_CIG101;
    break;

case TIF_CIG_TYPE_102:
    Sums.ClashCount++;
    TIFlog2ints ( 25, "%3ld --- %3ld <102> CONFLICT: differing ADDs",
        Second, First );
    nMsgId = TIF_STR_CIG102;
    break;

case TIF_CIG_TYPE_111:
    Sums.NoChgCount++;
    TIFlog3ints ( 25, "%3ld %3ld %3ld <111> Source+Target UNCHANGED",

```

```

        Third, First, Second );
    nMsgId = TIF_STR_CIG111;
    break;

case TIF_CIG_TYPE_112:
    Sums.ChgCount++;
    TIFlog3ints ( 25, "%3ld %3ld %3ld <112> Target CHG",
        Third, First, Second );
    nMsgId = TIF_STR_CIG112;
    break;

case TIF_CIG_TYPE_211:
    Sums.ChgCount++;
    TIFlog3ints ( 25, "%3ld %3ld %3ld <211> Source CHG",
        Third, First, Second );
    nMsgId = TIF_STR_CIG211;
    break;

case TIF_CIG_TYPE_212:
    Sums.ChgCount++;
    TIFlog3ints ( 25, "%3ld %3ld %3ld <212> identical SRC+TAR CHG",
        Third, First, Second );
    nMsgId = TIF_STR_CIG212;
    break;

case TIF_CIG_TYPE_213:
    Sums.ClashCount++;
    TIFlog3ints ( 25, "%3ld %3ld %3ld <213> conflicting SRC+TAR CHG",
        Third, First, Second );
    nMsgId = TIF_STR_CIG213;
    break;

case TIF_CIG_TYPE_132:
    Sums.ClashCount++;
    TIFlog3ints ( 25, "%3ld %3ld %3ld <132> ADDs or CHGs compromised",
        Third, First, Second );
    nMsgId = TIF_STR_CIG132;
    break;

//----- Never see 13F cuz we dump cig types before fanning starts!!
// case TIF_CIG_TYPE_13F:
//     Sums.ClashCount++;
//     TIFlog3ints ( 25, "%3ld %3ld %3ld <13F> 132 Fanned to Target",
//         Third, First, Second );
//     nMsgId = TIF_STR_CIG13F;
//     break;

default:
    TIFlog3ints ( 25, "CIG at %ld-->%ld has bad type %ld",
        First, Second, cigType );
    rc = TIF_ERR_BAD_CIG_TYPE;
}

if (mexType != 0)
{
    TIFlog1str ( 25, "          Changed to %s to Merge Exclusion Lists",
        TIFCigName (mexType) );

    //---- adjust resource ID used for logging to XLATE.LOG
    nMsgId += TIF_MEX_COUSIN_ADDEND;
}

if (rc != SUCCESS)
    return rc;

//---- Put entry in user-visible log, unless "muted"
if (nMsgId == TIF_NOTSET)
    return SUCCESS;
else
{
    rc = AddToXlateLog (pstTIF, First, nMsgId);
    return rc;
}
} //---- dump2_and_unmark_cig

```

```

/*-----
 * Name:      TIFDump2
 *
 * For all the world to see, log results of synchronization analysis in
 * the user-visible XLATE.LOG file.
 *
 * And for Debugging, put formatted dump of SYNCPORT conflicts in TIF.LOG.
 *-----*/
int TIFDump2 (PSTTIF_TYPE pstTIF)
{
    ILTR_PTRANSL tr = TIF_tr;
    ILDFX_PHNDL phFile = TIF_hFile;
    ILDFX_EC ec;
    INT32 lItemCount;
    INT32 i;
    INT32 lGroup = 0;
    char szCount[12];

    TIF_SUMS Sums;

    Sums.ClashCount = 0;
    Sums.ChgCount = 0;
    Sums.DelCount = 0;
    Sums.AddCount = 0;
    Sums.NoChgCount = 0;

    TIFlog0 (25, "\r\n=====");
    TIFlog0 (25, "=== IntelliSync Analysis of Corresponding Items ===");
    TIFlog0 (25, "=====\\r\\n");
    TIFlog0 (25, " SRC PRV TRG");
    TIFlog0 (25, " ### ### ### <SPT>");

    ec = ILDFX_GetRecordCount (phFile, &lItemCount);
    if (ec != ILDFX_OK)
        return ec;

    //----- Write header in user-visible XLATE.LOG
    ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_EOL, NULL, NULL);
    ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_EOL, NULL, NULL);
    ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_DIVIDER, NULL, NULL);
    ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_SUMMARY, NULL, NULL);

    //----- mark ALL records (we unmark as we dump)
    for (i=TIF_FIRST_ITEMNO; i < lItemCount; i++)
        TIFX_FLAGS2(phFile, i) |= (TIF2_MARK_DUMP1);

    //----- now dump all CIGs
    //-----TIF/ILDFX/IndexScanningLoop
    for (i=TIF_FIRST_ITEMNO; i < lItemCount; i++)
    {
        INT32 lFlags = TIFX_FLAGS(phFile, i);
        INT32 lFlags2 = TIFX_FLAGS2(phFile, i);

        //----- don't log zombies or wholly out-of-range items
        //----- or gobbled-up instances here
        if ( ((lFlags & (TIF_IS_GOBLED_UP_INSTANCE | TIF_IGNORE)) == 0)
            && ((lFlags2 & TIF2_WAS_OUTRANGE_10X) == 0)
            && (lFlags2 & TIF2_MARK_DUMP1) )
        {
            lGroup += 1;
            ec = dump2_and_unmark_cig (pstTIF, lGroup, i, Sums);
            if (ec != ILDFX_OK)
                return ec;
        }
    }

    //----- Write summary counts in user-visible XLATE.LOG
    ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_EOL, NULL, NULL);

    IL_SPRINTF (szCount, "%ld", Sums.ClashCount);
    ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_CLASH_COUNT, szCount, NULL);

    IL_SPRINTF (szCount, "%ld", Sums.AddCount);

```

```

ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_ADD_COUNT, szCount, NULL);

IL_SPRINTF (szCount, "%ld", Sums.ChgCount);
ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_CHG_COUNT, szCount, NULL);

IL_SPRINTF (szCount, "%ld", Sums.DelCount);
ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_DEL_COUNT, szCount, NULL);

IL_SPRINTF (szCount, "%ld", Sums.NoChgCount);
ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_NOCHG_COUNT, szCount, NULL);

ILAppendLog (ILTR_hLog, hXlatorInst, TIF_STR_DIVIDER, NULL, NULL);

return SUCCESS;

} //---- TIFDump2

/*-----
 * TIFCigName
 *-----*/
IL_PSTR TIFCigName (INT32 cigType)
{
    switch (cigType)
    {
        case TIF_CIG_TYPE_001: return ("ct001");
        case TIF_CIG_TYPE_100: return ("ct100");
        case TIF_CIG_TYPE_101: return ("ct101");
        case TIF_CIG_TYPE_102: return ("ct102");
        case TIF_CIG_TYPE_111: return ("ct111");
        case TIF_CIG_TYPE_112: return ("ct112");
        case TIF_CIG_TYPE_110: return ("ct110");
        case TIF_CIG_TYPE_211: return ("ct211");
        case TIF_CIG_TYPE_212: return ("ct212");
        case TIF_CIG_TYPE_213: return ("ct213");
        case TIF_CIG_TYPE_210: return ("ct210");
        case TIF_CIG_TYPE_011: return ("ct011");
        case TIF_CIG_TYPE_012: return ("ct012");
        case TIF_CIG_TYPE_010: return ("ct010");
        case TIF_CIG_TYPE_132: return ("ct132");
        case TIF_CIG_TYPE_13F: return ("ct13F");
        default:
        {
            static char szName[20];
            IL_SPRINTF (szName, "BAD_ct%lx", cigType);
            return szName;
        }
    }
}

} //---- TIFCigName

```

```

/*-----
* Name:      TIFMEX.CPP
* Part of:   the IntelliLink Synchronization Kernel ("TIF")
*
* Purpose:   mechanism for merging exclusion lists during synchronization
*           Not used at all for SmartMerge.
*
* Tables:    EMT1
*           EMT102
*           EMT213
*
* Functions (local functions indented):
*
*           TIFMergeExclusionLists
*           GetExclMergeInstructions
*           ExclMergeP
*
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1996
*-----*/

#include "iltr.h"

static int GetExclMergeInstructions ( PSTTIF_TYPE pstTIF,
                                     INT32 SItem,
                                     int ExclDelta,
                                     INT32 *pNewCigType,
                                     INT32 *pNewOutcome,
                                     int *pHowBits );

static int ExclMergeP ( PSTTIF_TYPE pstTIF,
                       int HowBits,
                       INT32 *MexList,
                       INT32 MexLen,
                       INT32 *pSItem,
                       INT32 *pTItem,
                       INT32 *pPItem );

//----- row numbers within a 3x3 table block
#define EXCL_DELTA_ONLY_T 0
#define EXCL_DELTA_ONLY_S 1
#define EXCL_DELTA_BOTH 2

//----- column numbers within a 3x3 table block
#define EXCL_NEWCIG_IDX 0
#define EXCL_NEWOUT_IDX 1
#define EXCL_HOW_IDX 2

//----- macros used to get entries from a row of an EMT table
#define EMT_NEWCIG(pRow) pRow[EXCL_NEWCIG_IDX]
#define EMT_NEWOUT(pRow) pRow[EXCL_NEWOUT_IDX]
#define EMT_HOW(pRow) pRow[EXCL_HOW_IDX]

//----- ERROR codes in tables defined with following macro
#define EMT_ERR(n) n

/*-----
* The 'PMEX' instructions tells us whether to create or modify the P-Item to
* contain the Merged Exclusion List and any other most current field values.
* We only do this when an UpdateBoth outcome is desired. CIG Type 132 is
* assigned to request UPDATE_BOTH.
*-----*/
#define EXCL_HOW_PMEX_NO 0x00
#define EXCL_HOW_PMEX_YES 0x01
#define EXCL_HOW_PMEX_T_ONLY 0x02
#define EXCL_HOW_PMEX_S_ONLY 0x04
#define EXCL_HOW_PMEX_TARG (EXCL_HOW_PMEX_T_ONLY | EXCL_HOW_PMEX_YES)
#define EXCL_HOW_PMEX_SOUR (EXCL_HOW_PMEX_S_ONLY | EXCL_HOW_PMEX_YES)

#define EXCL_HOW_MEX_ONLY 0x08
#define EXCL_HOW_ERROR 0x10
#define EXCL_HOW_WRONG_TABLE 0x20

/*-----
* EMT1 -- Excl Merge Table #1

```

```

*
* This table contains 9 entries per CIG type: 3 Rows X 3 Columns
*
* The columns are NewCigType, NewOutcome, and 'How'
* The rows are used as follows:
*
* row #0: when all unilateral exclusion ADDs are on the Target Side
* row #1: when all unilateral exclusion ADDs are on the Source Side
* row #2: when new exclusions are added on BOTH sides
*
*-----*/
static UINT8 EMT1 [TIFCIG_MAX+1] [3] [3] =
{
    //--- TIFCIG_000 (0) // no CIG type assigned

    EMT_ERR(11), 0, EXCL_HOW_ERROR,
    EMT_ERR(12), 0, EXCL_HOW_ERROR,
    EMT_ERR(13), 0, EXCL_HOW_ERROR,

    //--- TIFCIG_001 (1) // item is present in target only ("new in target")

    EMT_ERR(14), 0, EXCL_HOW_ERROR,
    EMT_ERR(15), 0, EXCL_HOW_ERROR,
    EMT_ERR(16), 0, EXCL_HOW_ERROR,

    //--- TIFCIG_100 (2) // item is present in source only ("new in source")

    EMT_ERR(17), 0, EXCL_HOW_ERROR,
    EMT_ERR(18), 0, EXCL_HOW_ERROR,
    EMT_ERR(19), 0, EXCL_HOW_ERROR,

    //--- TIFCIG_101 (3) // item is identical in Source and Target

    TIFCIG_102, TIFSYNC_TARGET_WINS, EXCL_HOW_PMAX_NO,
    TIFCIG_102, TIFSYNC_SOURCE_WINS, EXCL_HOW_PMAX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_YES,

    //--- TIFCIG_102 (4) // NEW SOURCE ITEM <> NEW TARGET ITEM

    EMT_ERR(20), 0, EXCL_HOW_WRONG_TABLE,
    EMT_ERR(21), 0, EXCL_HOW_WRONG_TABLE,
    EMT_ERR(22), 0, EXCL_HOW_WRONG_TABLE,

    //--- TIFCIG_111 (5) // item is unchanged across the board

    TIFCIG_213, TIFSYNC_TARGET_WINS, EXCL_HOW_PMAX_NO,
    TIFCIG_213, TIFSYNC_SOURCE_WINS, EXCL_HOW_PMAX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_YES,

    //--- TIFCIG_112 (6) // item CHANGED in Target since last sync

    TIFCIG_112, 0, EXCL_HOW_PMAX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_TARG,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_TARG,

    //--- TIFCIG_110 (7) // item DELETED from Target since last sync

    EMT_ERR(26), 0, EXCL_HOW_ERROR,
    EMT_ERR(27), 0, EXCL_HOW_ERROR,
    EMT_ERR(28), 0, EXCL_HOW_ERROR,

    //--- TIFCIG_211 (8) // item CHANGED in Source since last sync

    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_SOUR,
    TIFCIG_211, 0, EXCL_HOW_PMAX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_SOUR,

    //--- TIFCIG_212 (9) // item CHANGED IDENTICALLY in Src & Target

    TIFCIG_213, TIFSYNC_TARGET_WINS, EXCL_HOW_PMAX_NO,
    TIFCIG_213, TIFSYNC_SOURCE_WINS, EXCL_HOW_PMAX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMAX_SOUR,

    //--- TIFCIG_213 (10) // item CHANGED DIFFERENTLY in Src & Target

```



```

        EMT_ERR(29), 0,                EXCL_HOW_WRONG_TABLE,
        EMT_ERR(30), 0,                EXCL_HOW_WRONG_TABLE,
        EMT_ERR(31), 0,                EXCL_HOW_WRONG_TABLE,

//--- TIFCIG_210 (11) // item CHANGED in Source, DELETED from Target

        EMT_ERR(32), 0,                EXCL_HOW_ERROR,
        EMT_ERR(33), 0,                EXCL_HOW_ERROR,
        EMT_ERR(34), 0,                EXCL_HOW_ERROR,

//--- TIFCIG_011 (12) // item DELETED from Source since last sync

        EMT_ERR(35), 0,                EXCL_HOW_ERROR,
        EMT_ERR(36), 0,                EXCL_HOW_ERROR,
        EMT_ERR(37), 0,                EXCL_HOW_ERROR,

//--- TIFCIG_012 (13) // item DELETED from Source, CHANGED in Target,

        EMT_ERR(38), 0,                EXCL_HOW_ERROR,
        EMT_ERR(39), 0,                EXCL_HOW_ERROR,
        EMT_ERR(40), 0,                EXCL_HOW_ERROR,

//--- TIFCIG_010 (14) // item DELETED from both Source & Target

        EMT_ERR(41), 0,                EXCL_HOW_ERROR,
        EMT_ERR(42), 0,                EXCL_HOW_ERROR,
        EMT_ERR(43), 0,                EXCL_HOW_ERROR,

//--- TIFCIG_132 (15) // 102 conflict resolved interactively
// to a "compromise" value stored in P-item

        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_YES,
        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_YES,
        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_YES

}; //---- EMT1

/*-----
 * EMT102 -- Excl Merge Table used for CIGs of type 102
 *
 * Similar to the EMT1 table, but instead of 9 entries per CIG type, this
 * table has 9 entries per outcome, for a ADD-ADD conflict CIG.
 *-----*/
static UINT8 EMT102 [TIF_SYNC_OUTCOME_COUNT+1] [3] [3] =
{

//--- Outcome not set (0)

        EMT_ERR(47), 0,                EXCL_HOW_ERROR,
        EMT_ERR(48), 0,                EXCL_HOW_ERROR,
        EMT_ERR(49), 0,                EXCL_HOW_ERROR,

//--- Target Wins (1)

        TIFCIG_102, TIFSYNC_TARGET_WINS, EXCL_HOW_PMEX_NO,
        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_TARG,
        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_TARG,

//--- Source Wins (2)

        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_SOUR,
        TIFCIG_102, TIFSYNC_SOURCE_WINS, EXCL_HOW_PMEX_NO,
        TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_SOUR,

//--- Ignore (3)
//--- Here we have an un-resolved conflict between two
//--- not-previously-synchronized items. Do Not Merge Exclusions!!

        TIFCIG_102, TIFSYNC_IGNORE,      EXCL_HOW_PMEX_NO,
        TIFCIG_102, TIFSYNC_IGNORE,      EXCL_HOW_PMEX_NO,
        TIFCIG_102, TIFSYNC_IGNORE,      EXCL_HOW_PMEX_NO

}; //---- EMT102

```

```

/*-----
 * EMT213 -- Excl Merge Table used for CIGs of type 213
 *
 * Similar to the EMT1 table, but instead of 9 entries per CIG type, this
 * table has 9 entries per outcome, for a CHG-CHG conflict CIG.
 *-----*/
static UINT8 EMT213 [TIF_SYNC_OUTCOME_COUNT+1] [3] [3] =
{
    //--- Outcome not set (0)

    EMT_ERR(53), 0, EXCL_HOW_ERROR,
    EMT_ERR(54), 0, EXCL_HOW_ERROR,
    EMT_ERR(55), 0, EXCL_HOW_ERROR,

    //--- Target Wins (1)

    TIFCIG_213, TIFSYNC_TARGET_WINS, EXCL_HOW_PMEX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_TARG,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_TARG,

    //--- Source Wins (2)

    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_SOUR,
    TIFCIG_213, TIFSYNC_SOURCE_WINS, EXCL_HOW_PMEX_NO,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_SOUR,

    //--- Ignore (3)

    TIFCIG_213, TIFSYNC_TARGET_WINS, EXCL_HOW_PMEX_YES | EXCL_HOW_MEX_ONLY,
    TIFCIG_213, TIFSYNC_SOURCE_WINS, EXCL_HOW_PMEX_YES | EXCL_HOW_MEX_ONLY,
    TIFCIG_132, 0 /* UpdateBoth */, EXCL_HOW_PMEX_YES | EXCL_HOW_MEX_ONLY
}; //---- EMT213

/*-----
 * Name: TIFMergeExclusionLists -- called from set_cig_type in TIFSYNC.CPP
 *
 * NOTE: this function assumes that all exclusion lists are fully sanitized
 * (sorted into increasing order, with no duplicate entries)
 *-----*/
int TIFMergeExclusionLists ( PSTTIF_TYPE pstTIF, INT32 FirstItem,
                           INT32 SItem, INT32 TItem, INT32 PItem )
{
    ILTR_PDATES SexList;
    ILTR_PDATES TexList;
    ILTR_PDATES MexList;
    INT32 SexLen;
    INT32 TexLen;
    INT32 MexLen;
    INT32 si, ti, mi;
    int SourceOnlyCount;
    int TargetOnlyCount;
    int ExclDelta;
    IL_PSTR lpszDelta;
    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 OldCigType = TIFX_CIG_TYPE(phFile, FirstItem);
    INT32 NewCigType;
    INT32 NewOutcome;
    int HowBits;
    int rc;
    BOOLEAN bItemIsRecurring = TIFX_ITEM_IS_RECURRING (phFile, FirstItem);
    int fieldnum = TIF_RepExclFieldNum;

    //---- if item isn't recurring this function is a NO-OP
    if (!bItemIsRecurring)
        return SUCCESS;

    /*-----
 * For us to do an exclusion list merge we need to have both Source and
 * Target Items. (Note that cig type 132 has a deviant usage for the
 * so-called Previous Item. CIG type 213 may also (when outcome is

```

```

* Update Both). But for these cases the exclusion list of the
* so-called P-Item is to be ignored. So we always pay attention to the
* Source and Target Exclusion Lists.
*-----*/
switch (OldCigType)
{
    case TIF_CIG_TYPE_001:
    case TIF_CIG_TYPE_100:
    case TIF_CIG_TYPE_110:
    case TIF_CIG_TYPE_210:
    case TIF_CIG_TYPE_011:
    case TIF_CIG_TYPE_012:
    case TIF_CIG_TYPE_010: return SUCCESS;

    case TIF_CIG_TYPE_101:
    case TIF_CIG_TYPE_102:
    case TIF_CIG_TYPE_111:
    case TIF_CIG_TYPE_112:
    case TIF_CIG_TYPE_211:
    case TIF_CIG_TYPE_212:
    case TIF_CIG_TYPE_213:
    case TIF_CIG_TYPE_132: break;

    default: return ILERROR_L (OldCigType, TIF_ERR_ABNORMAL);
}

//---- NO-OP if source and target have identical exclusion lists
if (TIFX_REP_EXCL_HASH(phFile, SItem) == TIFX_REP_EXCL_HASH(phFile, TItem))
    return SUCCESS;

//---- read the Source Item into 'First Record'
rc = TIFRetrieveRecord (pstTIF, SItem, &TIF_FirstRecord);
if (rc != SUCCESS) return (ILERROR_L (SItem, rc));

//---- read the Target Item into 'Second Record'
rc = TIFRetrieveRecord (pstTIF, TItem, &TIF_SecondRecord);
if (rc != SUCCESS) return (ILERROR_L (TItem, rc));

//---- locate the Exclusion Lists
SexList = (ILTR_PDATES) TIF_FieldData(TIF_pFirstRecord, fieldnum);
TexList = (ILTR_PDATES) TIF_FieldData(TIF_pSecondRecord, fieldnum);

//---- get List Lengths (in bytes)
SexLen = TIF_FieldLength(TIF_pFirstRecord, fieldnum);
TexLen = TIF_FieldLength(TIF_pSecondRecord, fieldnum);

//---- make sure CurrentField buffer is big enough for merge
rc = ILUT_GetBuffer (&TIF_CurrentField, SexLen + TexLen);
if (rc != SUCCESS) return (ILERROR_L (SexLen + TexLen, rc));

MexList = (ILTR_PDATES) TIF_pCurrentField;

//---- change List Lengths to be counts of list entries
SexLen /= sizeof(long);
TexLen /= sizeof(long);

//---- initialize counts of exclusions that are NOT common to both lists
SourceOnlyCount = 0;
TargetOnlyCount = 0;

//---- initialize indices for traversing lists (source, target, merge)
si = 0;
ti = 0;
mi = 0;

//---- build Merged List, determine its length, and count unilateral inputs
while (si < SexLen && ti < TexLen)
{
    //---- copy Source entries that precede OR MATCH the next target entry
    while (si < SexLen && ti < TexLen && SexList[si] <= TexList[ti])
    {
        MexList[mi++] = SexList[si];
        if (SexList[si] == TexList[ti])
            ti++;
        else

```

```

        SourceOnlyCount++;
        si++;
    }

    //---- copy all Target entries that precede the next Source entry
    while (si < SexLen && ti < TexLen && TexList[ti] < SexList[si])
    {
        MexList[mi++] = TexList[ti++];
        TargetOnlyCount++;
    }

    //---- if there are leftover Source entries, copy them across
    while (si < SexLen)
    {
        MexList[mi++] = SexList[si++];
        SourceOnlyCount++;
    }

    //---- if there are leftover Target entries, copy them across
    while (ti < TexLen)
    {
        MexList[mi++] = TexList[ti++];
        TargetOnlyCount++;
    }

    //---- note length of merged list. Zero length is bad.
    MexLen = mi;
    TIFlog3ints ( 70, "Merged Excl Lists: %ld + %ld = %ld",
                  SexLen, TexLen, MexLen );
    if (MexLen == 0)
        return ILERROR (0, TIF_ERR_ABNORMAL);

    //---- something is amiss if the two lists are identical!!
    if (SourceOnlyCount == 0 && TargetOnlyCount == 0)
        return ILERROR (0, TIF_ERR_ABNORMAL);

    //---- Characterize the overall set of Exclusion List changes
    if (SourceOnlyCount == 0)
    {
        ExclDelta = EXCL_DELTA_ONLY_T;
        lpszDelta = "Target";
    }
    else if (TargetOnlyCount == 0)
    {
        ExclDelta = EXCL_DELTA_ONLY_S;
        lpszDelta = "Source";
    }
    else
    {
        ExclDelta = EXCL_DELTA_BOTH;
        lpszDelta = "Both";
    }

    //---- Look in table to see how exclusion merge affects record outcome
    rc = GetExclMergeInstructions ( pstTIF, SItem, ExclDelta,
                                   &NewCigType, &NewOutcome, &HowBits );
    if (ILLOG_VERBOSE_ENOUGH(TIFLOG, 70))
    {
        char szLog[100];
        IL_SPRINTF ( szLog, "ExclMerge (%s, delta=%s) ==> %s %lx %x; rc=%d",
                     TIFCigName(OldCigType), lpszDelta, TIFCigName(NewCigType),
                     NewOutcome, HowBits, rc );
        TIFlogsz (szLog);
    }

    if (rc != SUCCESS)
        return ILERROR (rc, TIF_ERR_ABNORMAL);

    //---- If necessary put merged exclusion list into the P-Item. This will
    //---- also cause a P-Item to be created if none exists yet (i.e. for
    //---- CIG types 101 and 102.)
    if (HowBits & EXCL_HOW_PMEY_YES)
    {
        rc = ExclMergeP ( pstTIF, HowBits, MexList, MexLen,

```

```

        &SItem, &TItem, &PItem );
    if (rc != SUCCESS)
        return ILERROR (rc, rc);
}

//---- Get ready to make changes to the FLAGS words for all CIG+FIG members
INT32 NewFlags;
INT32 NewFlags2;

//---- Always put the new CIG type and outcome in FLAGS
NewFlags = NewCigType | NewOutcome;

//---- If CIG type is unchanged, do nothing to FLAGS2
if (NewCigType == OldCigType)
    NewFlags2 = 0;

//---- If CIG type is changing, set bit and store old CIG type in FLAGS2
else
{
    NewFlags |= TIF_MEX_CIG;
    NewFlags2 = OldCigType;
}

//---- Set 'Mex Only' flag if non-Excl conflict is to remain unresolved
if (HowBits & EXCL_HOW_MEX_ONLY)
    NewFlags |= TIF_MEX_ONLY;

//---- Assign new cig type and new outcome to all affected records
rc = TIFMarkAllFigMembers ( phFile, SItem,
                           TIF_CIG_TYPE_MASK | TIF_OUTCOME_MASK,
                           NewFlags, NewFlags2 );

if (rc != SUCCESS)
    return ILERROR_L (SItem, rc);

rc = TIFMarkAllFigMembers ( phFile, TItem,
                           TIF_CIG_TYPE_MASK | TIF_OUTCOME_MASK,
                           NewFlags, NewFlags2 );

if (rc != SUCCESS)
    return ILERROR_L (TItem, rc);

if (PItem != TIF_NOTSET)
{
    rc = TIFMarkAllFigMembers ( phFile, PItem,
                               TIF_CIG_TYPE_MASK | TIF_OUTCOME_MASK,
                               NewFlags, NewFlags2 );

    if (rc != SUCCESS)
        return ILERROR_L (PItem, rc);
}

return SUCCESS;
} //---- TIFMergeExclusionLists

/*-----
* Name:      GetExclMergeInstructions -- called from MergeExclusionLists
*
* Determine how Exclusion List merge shall affect overall record sync.
*-----*/
static int GetExclMergeInstructions ( PSTTIF_TYPE pstTIF,
                                     INT32 SItem,
                                     int ExclDelta,
                                     INT32 *pNewCigType,
                                     INT32 *pNewOutcome,
                                     int *pHowBits )
{
    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 flags = TIFX_FLAGS(phFile, SItem);
    INT32 cigType = flags & TIF_CIG_TYPE_MASK;
    INT32 cigNum = cigType >> TIF_CIG_TYPE_SHIFT;
    INT32 outcome = flags & TIF_OUTCOME_MASK;
    INT32 outNum = outcome >> TIF_OUTCOME_SHIFT;

    INT32 newCigNum;
    INT32 newOutNum;

```

```

UINT8 *pRow; // ptr to a row in an EMT table

//---- sanity check the cig num
if (cigNum > TIFCIG_MAX)
    return ILERROR_L (cigNum, TIF_ERR_ABNORMAL);

//---- sanity check the outcome num
if (outNum > TIF_SYNC_OUTCOME_COUNT)
    return ILERROR_L (cigNum, TIF_ERR_ABNORMAL);

//---- locate row in appropriate EMT table
switch (cigNum)
{
    case TIFCIG_102: pRow = &EMT102 [outNum][ExclDelta][0]; break;
    case TIFCIG_213: pRow = &EMT213 [outNum][ExclDelta][0]; break;
    default:        pRow = &EMT1   [cigNum][ExclDelta][0]; break;
}

newCigNum = (INT32) EMT_NEWCIG(pRow);
*pNewCigType = newCigNum << TIF_CIG_TYPE_SHIFT;

newOutNum = (INT32) EMT_NEWOUT(pRow);
*pNewOutcome = newOutNum << TIF_OUTCOME_SHIFT;

*pHowBits = (int) EMT_HOW(pRow);
if (*pHowBits & (EXCL_HOW_ERROR | EXCL_HOW_WRONG_TABLE))
    return ILERROR_L (cigNum, (int) newCigNum);

return SUCCESS;
} //---- GetExclMergeInstructions

/*-----
 * Name:      ExclMergeP -- called from MergeExclusionLists
 *
 * This function creates or modifies a CIG's P-Item. At minimum this
 * function puts the Merged Exclusion List into the P-Item and fixes the
 * exclusion count in RepBasic. In addition it may create the P-item from
 * scratch, or replace a pre-existing P-Item, by copying the Source or
 * Target Item.
 *-----*/
static int ExclMergeP ( PSTTIF_TYPE pstTIF,
                      int HowBits,
                      INT32 *MexList,
                      INT32 MexLen,
                      INT32 *pSItem,
                      INT32 *pTItem,
                      INT32 *pPItem )
{
    ILTR_PREPEAT pRepeat;
    INT32 len;
    int fldnum;
    INT32 NewItem;
    int rc;
    ILDFX_PHNDL phFile = TIF_hFile;
    TIF_RECORD_VALUE_PTR pReplacementRec = NULL;

    if (*pPItem == TIF_NOTSET)
    {
        if (HowBits & EXCL_HOW_PMAX_S_ONLY)
        {
            rc = TIFCommandeerSourceItem ( pstTIF, *pSItem, *pTItem,
                                           TIF_pFirstRecord, &NewItem );
            if (rc != SUCCESS) return rc;

            *pPItem = *pSItem;
            *pSItem = NewItem;
        }

        //---- Copy Target if not told to copy source.
        else
        {
            rc = TIFCommandeerTargetItem ( pstTIF, *pTItem, *pSItem,

```

```

        TIF_pSecondRecord, &NewItem );
    if (rc != SUCCESS) return rc;

    *pPItem = *pTItem;
    *pTItem = NewItem;
}

else if (HowBits & EXCL_HOW_PMAX_S_ONLY)
    pReplacementRec = TIF_pFirstRecord;

else if (HowBits & EXCL_HOW_PMAX_T_ONLY)
    pReplacementRec = TIF_pSecondRecord;

//---- Replace previous P-Item with Source or Target, if so directed
if (pReplacementRec != NULL)
{
    rc = ILDFX_UpdateRecord ( phFile, *pPItem,
                             pReplacementRec,
                             TIFREC_SIZE(pReplacementRec), NULL );

    if (rc != ILDFX_OK)
        return ILERROR (rc, TIF_ERR_ABNORMAL);
}

//---- We always install the merged exclusion list in the P-Item.
//---- Start by reading the P-Item into 'Current Record'.
rc = TIFRetrieveRecord (pstTIF, *pPItem, &TIF_CurrentRecord);
if (rc != SUCCESS) return ILERROR_L (*pPItem, rc);

//---- Put the _repExcl field value into Current Record
fldnum = TIF_RepExclFieldNum;
rc = TIFPutFieldByIndex ( pstTIF, fldnum,
                          MexList, MexLen * sizeof(long) );
if (rc != SUCCESS) return ILERROR (rc, rc);

//---- Make sure we have a valid _repBasic field to update
fldnum = TIF_RepBasicFieldNum;
len = TIF_FieldLength(TIF_pCurrentRecord, fldnum);
if (len != sizeof(ILTR_REPEAT))
    return ILERROR_L (len, TIF_ERR_ABNORMAL);

//---- Update the exclusion count in the _repBasic field
pRepeat = (ILTR_REPEAT) TIF_FieldData(TIF_pCurrentRecord, fldnum);
pRepeat->numExDates = (INT16) MexLen;

//---- Get ready to compute new EXDATA values for P-Item
INT32 exdata[TIF_EXDATA_PER_RECORD];
IL_MEMSET((IL_PANY) exdata, 0, sizeof(exdata));

//---- retain pre-existing Flags and Group Memberships
exdata[TIF_FLAGS_SLOT] = TIFX_FLAGS(phFile, *pPItem);
exdata[TIF_FLAGS2_SLOT] = TIFX_FLAGS2(phFile, *pPItem);
exdata[TIF_NEXT_IN_CIG_SLOT] = TIFX_NEXT_IN_CIG(phFile, *pPItem);
exdata[TIF_NEXT_IN_SKG_SLOT] = TIFX_NEXT_IN_SKG(phFile, *pPItem);
exdata[TIF_NEXT_IN_FIG_SLOT] = TIFX_NEXT_IN_FIG(phFile, *pPItem);

//---- Re-compute everything else (hash values and start/end DTTM values)
rc = TIFComputeSearchKeyValues(pstTIF, TIF_pCurrentRecord, exdata);
if (rc != SUCCESS)
    return ILERROR (rc, TIF_ERR_ABNORMAL);

//---- Update the P-Item record on disk
rc = ILDFX_UpdateRecord ( phFile, *pPItem,
                          TIF_pCurrentRecord,
                          TIFREC_SIZE(TIF_pCurrentRecord), exdata );

if (rc != ILDFX_OK)
    return ILERROR (rc, TIF_ERR_ABNORMAL);

return SUCCESS;
} //---- ExclMergeP

```

```

/*-----
* Name:      TIFPUT.CPP
*
* Contains functions (local functions indented):
*
*   TIFPutRecord:  used when loading TIF for SmartMerge or SyncPort
*       LogNonUniqueID
*       CheckFastSyncDelta
*       StoreUnanalyzedRecord
*       AnalyzeAndStoreRecord
*   TIFAdjustRecordIfChanged:  used in aftermath of reconciliation
*       VerifyIDMatch
*       SearchForIDMatch
*       SearchForKeyFieldsMatch
*       EffectiveFieldLength
*       InitiateNewGroupMember
*   TIFSetSimpleApptSpan
*       SetRecurringApptSpan
*       SetSimpleTodoSpan
*       SetRecurringTodoSpan
*       DigestRecurrencePattern
*   TIFComputeSearchKeyValues
*       DateCompare
*       SanitizeExclusionList
*   TIFSanitizeRepBasic
*       CheckAndSetOutOfRange
*       IsRecurringOutOfRange
*       IsSimpleOutOfRange
*
* Different flavors of PutRecord are:
*
* 1.  analyze and store a new record.  Analysis is done to compute all
*     EXDATA values, which includes establishing CIG & SKG membership ties.
*
* 2.  store a record without doing any analysis.
*
* 3.  update a record that was previously stored w/o analysis (by 1).
*     the record may have been altered since it was first stored.
*     This same update operation does analysis to set all EXDATA values.
*
* 4.  special-purpose exdata-only update done to assimilate a history file.
*
* 5.  adjust a previously analyzed and stored record, whose ID is guaranteed
*     NOT to have changed, but whose other field values, including KeyField
*     and DTTM values, may have changed.  Hence SKG membership may shift.
*
* Flavors 1-4 are all accomplished by calling 'TIFPutRecord'.
* Flavor 5 is done by calling 'TIFAdjustRecordIfChanged'.
*
* For flavor #2, TIFPutRecord calls StoreUnanalyzedRecord.
* For the other flavors, TIFPutRecord calls AnalyzeAndStoreRecord.
*
* Author:  David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/

```

```
#include "iltr.h"
```

```

static int LogNonUniqueID ( ILTR_PTRANSL tr,
                           PSTTIF_TYPE pstTIF,
                           TIF_RECORD_VALUE_PTR pRec,
                           int idError );

static int CheckFastSyncDelta ( PSTTIF_TYPE pstTIF,
                                TIF_RECORD_VALUE_PTR pRecord,
                                BOOLEAN IL_DIST *pisaZombie );

static int StoreUnanalyzedRecord ( ILTR_PTRANSL tr,
                                   TIF_RECORD_VALUE_PTR pRec );

static int AnalyzeAndStoreRecord ( ILTR_PTRANSL tr,
                                   ILUT_PBUFFER pRecBuf,
                                   TIF_PUT_RECORD_OPTION nPutOption,
                                   BOOLEAN isaZombie,
                                   INT32 ItemToIgnore,
                                   BOOLEAN *pMustAddZombie );

```



```

static int SearchForIDMatch ( PSTTIF_TYPE pstTIF,
                             TIF_RECORD_VALUE_PTR pRec,
                             INT32 *exdata,           // in (array)
                             int slotIndex,
                             INT32 ExcludedItem1,
                             INT32 ExcludedItem2,
                             INT32 LastPossibleMatch,
                             INT32 *pMatchIndex );    // out

static int SearchForKeyFieldsMatch ( PSTTIF_TYPE pstTIF,
                                     TIF_RECORD_VALUE_PTR pRec,
                                     INT32 lHashSought,
                                     INT32 ExcludedItem,
                                     INT32 LastPossibleMatch,
                                     INT32 *pMatchIndex );

static int InitiateNewGroupMember ( ILDFX_PHNDL phFile,
                                    INT32 RecordNumber,
                                    int LinkSlot,
                                    IL_PSTR szGroupType );

static int SanitizeExclusionList ( PSTTIF_TYPE pstTIF,
                                  TIF_RECORD_VALUE_PTR pRecord,
                                  ILTR_PREPEAT pRepeat );

static int CheckAndSetOutOfRange ( ILTR_PTRANSL tr,
                                   PSTTIF_TYPE pstTIF,
                                   TIF_RECORD_VALUE_PTR pRec,
                                   INT32 CurrentItem,
                                   INT32 *exdata );

static BOOLEAN IsRecurringOutOfRange ( ILTR_PTRANSL tr,
                                       PSTTIF_TYPE pstTIF,
                                       TIF_DATERANGE Range,
                                       TIF_RECORD_VALUE_PTR pRec,
                                       ILTR_PREPEAT pRepeat );

static BOOLEAN IsSimpleOutOfRange ( PSTTIF_TYPE pstTIF,
                                    TIF_DATERANGE Range,
                                    long lStartDate,
                                    long lEndDate );

extern "C" int IL_CDECL DateCompare ( const void *pKey_1, const void *pKey_2 );

/*-----
 * Name:      TIFPutRecord
 *
 * Purpose:  Optionally Analyze and Store a record.
 *
 * Details:  Depending on the "Put Option" parameter, one of the following
 *           operations is performed:
 *
 *   1. just store a new record, unanalyzed, or
 *   2. analyze record to set CIG & SKG membership, then store it as a new
 *      record, or update an previously stored record.
 *-----*/
int TIFPutRecord (ILTR_PTRANSL tr, TIF_PUT_RECORD_OPTION nPutOption)
{
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

    //---- use the Current Record Buffer (so that pRec == TIF_pCurrentRecord)
    ILUT_PBUFFER pRecBuf = &TIF_CurrentRecord;
    TIF_RECORD_VALUE_PTR pRec = (TIF_RECORD_VALUE_PTR) pRecBuf->pBuffer;

    int rc = SUCCESS;
    BOOLEAN isaZombie = FALSE;    // TRUE for FastSync DELETE items
    BOOLEAN bMustAddZombie = FALSE; // TRUE after ILLEGAL MATCH for FastSync
    INT32 ItemToIgnore;

    /*-----
     * When doing Fast Sync Load validate the _Delta field and detect zombies.
     *-----*/
    if (TIF_bFastSyncLoad)
        rc = CheckFastSyncDelta (pstTIF, pRec, &isaZombie);
}

```

```

if (rc == SUCCESS)
{
    switch (nPutOption)
    {
        case TIFPRO_STORE_NEW_UNANALYZED_RECORD:

            /*-----
            * Most records are NOT analyzed here, but FastSync zombies ARE
            * analyzed here, cuz we don't want the Target Translator to try
            * to SANITIZE zombies. They look pretty deviant, so translators
            * are likely to dislike them.
            *-----*/
            if (isaZombie)
                // change option and fall through into next case
                nPutOption = TIFPRO_ANALYZE_AND_STORE_NEW_RECORD;
            else
            {
                rc = StoreUnanalyzedRecord(tr, pRec);
                break;
            }

        case TIFPRO_ANALYZE_AND_STORE_NEW_RECORD:
        case TIFPRO_ANALYZE_AND_UPDATE_RECORD:
        case TIFPRO_SPECIAL_HISTORY_FILE_UPDATE:

            ItemToIgnore = -1;
            rc = AnalyzeAndStoreRecord( tr, pRecBuf, nPutOption, isaZombie,
                                      ItemToIgnore, &bMustAddZombie );
            break;

        default:

            rc = TIF_ERR_BAD_PUT_RECORD_OPTION;
            break;
    }
}

if (rc == SUCCESS)
    //----- set special action code to suppress logging by IMPORT.C
    ILSetAction ( tr, ILTR_ACT_LOADED_INTO_TIF );

ILTIFlogszul3 ( "TIFPutRecord/%ld (item #%ld) ==> rc=%ld",
                (UINT32) nPutOption,
                (UINT32) TIF_lCurrentRecNum,
                (UINT32) rc );

/*-----
* Fault tolerance: don't croak if translator supplies non-unique IDs.
*-----*/
if (rc == TIF_ERR_NON_UNIQUE_SOURCEID || rc == TIF_ERR_NON_UNIQUE_TARGETID)
{
    //---- put a stern warning in the user-visible logfile
    rc = LogNonUniqueID (tr, pstTIF, pRec, rc);

    // Get ready for the next record by making sure it will be primed
    TIF_bNeedPriming = TRUE;
    return SUCCESS;
}

/*-----
* For Fast Sync, we have thus far ensured that an Illegal Match will
* generate an ADD coming from the FastSync side. Now we add a zombie
* so that there will also be a DELETE coming from the FastSync side.
*-----*/
if (rc == SUCCESS && bMustAddZombie)
{
    INT16 fieldnum;
    char szDelta[2];

    if (TIF_phase == TIF_PHASE_LOADING_TARGET_RECORDS)
        fieldnum = TIF_TargetDeltaFieldNum;
    else
        fieldnum = TIF_SourceDeltaFieldNum;
}

```

```

//---- save a copy of the original DELTA code
szDelta[0] = (TIF_FieldData(pRec, fieldnum))[0];
szDelta[1] = 0;

//---- change the DELTA code to DELETE
rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList, "", fieldnum,
                             ILTR_DELTA_DELETE, 0, FALSE, pRecBuf );
if (rc != SUCCESS) return ILERROR (rc, rc);

isaZombie = TRUE;
nPutOption = TIFPRO_STORE_SPECIAL_ZOMBIE;
ItemToIgnore = TIF_lCurrentRecNum;

//---- put out a DELETE for the same record we just added
rc = AnalyzeAndStoreRecord ( tr, pRecBuf, nPutOption, isaZombie,
                             ItemToIgnore, &bMustAddZombie );

ILTIFlogszul3 ( "PutSpecialDelete/%ld (item %ld) ==> rc=%ld",
                (UINT32) nPutOption,
                (UINT32) TIF_lCurrentRecNum,
                (UINT32) rc );

if (rc != SUCCESS) return ILERROR (rc, rc);

//---- restore the original DELTA code
rc = TIFRecordAddFieldValue ( pstTIF, TIF_pFieldList, "", fieldnum,
                             szDelta, 0, FALSE, pRecBuf );
if (rc != SUCCESS) return ILERROR (rc, rc);
}

// Get ready for the next record by making sure it will be primed
TIF_bNeedPriming = TRUE;

return rc;
} //---- TIFPutRecord

/*-----
 * LogNonUniqueID
 *-----*/
static int LogNonUniqueID ( ILTR_PTRANS� tr,
                           PSTTIF_TYPE pstTIF,
                           TIF_RECORD_VALUE_PTR pRec,
                           int idError )
{
    int rc;
    int fieldnum = TIF_ViewFieldNum;
    IL_PSTR psz;

    if (TIF_FieldLength(pRec, fieldnum) == 0)
        LoadString( TIF_DLL_InstanceHandle, TIF_STR_UNSPECIFIED,
                    ILTR_szRecName, MAX_MSG );
    else
    {
        psz = TIF_FieldData(pRec, fieldnum);
        IL_SAFE_STRINGCOPY (ILTR_szRecName, psz);
        //---- Truncate the name at EOS char (end of line)
        IL_PSTR lpMatch = IL_STRCHR (ILTR_szRecName, ILTR_EOS_CHAR);
        if (lpMatch != NULL)
            *lpMatch = 0;
    }

    //----- Write out log record, saying that record is being IGNORED
    rc = ILAppendLog ( ILTR_hLog, ILTR_hRes, ILTR_MSG_IGNORE,
                      ILTR_szRecName, NULL );
    if (rc != SUCCESS)
        return ILTR_ERR_LOGFILE;

    //---- Get the Unique ID string
    if (idError == TIF_ERR_NON_UNIQUE_SOURCEID)
        fieldnum = TIF_SourceIDFieldNum;
    else
        fieldnum = TIF_TargetIDFieldNum;
}

```

```

    psz = TIF_FieldData(pRec, fieldnum);

    char szTemplate[80];
    char szLogEntry[300];

    //---- Get template for non-unique ID explanation
    LoadString( TIF_DLL_InstanceHandle, TIF_STR_NON_UNIQUE_ID,
                szTemplate, sizeof(szTemplate) );

    //---- Explain why we're ignoring this record
    IL_SPRINTF (szLogEntry, szTemplate, psz);
    IL_WRITE (ILTR_hLog, szLogEntry, IL_STRLEN(szLogEntry), rc);

    return SUCCESS;
} //---- LogNonUniqueID

/*-----
 * Name:      CheckFastSyncDelta
 * Called from TIFPutRecord, only during FastSync loading of Target or Source
 * Records, to validate the _Delta field value and pass back the "isaZombie"
 * indication if _Delta=D (Delete).
 *-----*/
static int CheckFastSyncDelta ( PSTTIF_TYPE pstTIF,
                                TIF_RECORD_VALUE_PTR pRecord,
                                BOOLEAN IL_DIST *pisaZombie )
{
    int fieldnum;
    INT32 len;
    char cDelta;

    //---- choose _Delta field based on current TIF phase
    if (TIF_phase == TIF_PHASE_LOADING_TARGET_RECORDS)
        fieldnum = TIF_TargetDeltaFieldNum;
    else if (TIF_phase == TIF_PHASE_LOADING_SOURCE_RECORDS)
        fieldnum = TIF_SourceDeltaFieldNum;
    else if (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
        fieldnum = TIF_SourceDeltaFieldNum;
    else
        return ILERROR (TIF_phase, TIF_ERR_BAD_STATE);

    //---- When doing FastSync Load there had better be a _Delta field
    if (fieldnum == TIF_NOTSET)
        return ILERROR (fieldnum, TIF_ERR_ABNORMAL);

    //---- All legal _Delta field values are 1 letter followed by a NULL
    len = TIF_FieldLength(pRecord, fieldnum);
    if (len != 2)
        return ILERROR_L (len, TIF_ERR_BAD_FASTSYNC_DELTA);

    //---- Grab the 1-character _Delta value
    cDelta = (TIF_FieldData(pRecord, fieldnum))[0];

    switch (cDelta)
    {
        case ILTR_CDELTA_ADD:
        case ILTR_CDELTA_CHANGE:
            *pisaZombie = FALSE;
            return SUCCESS;

        case ILTR_CDELTA_DELETE:
            *pisaZombie = TRUE;
            return SUCCESS;

        default:
            return ILERROR ((int) cDelta, TIF_ERR_BAD_FASTSYNC_DELTA);
    }
} //---- CheckFastSyncDelta

/*-----

```

```

* Name:      StoreUnanalyzedRecord (called from TIFPutRecord)
*
* Purpose: Simply Store a record. This is done when Source Translator is
*           putting records into TIF. Analysis is deferred until a later
*           phase of translation, at which point the Target Translator gets
*           a chance to sanitize the Source Records. At that point the
*           TIFPRO_ANALYZE_AND_UPDATE_RECORD option is typically used, unless
*           the Target Translator needs to do FANNING or other record-creating
*           operations.
*-----*/
static int StoreUnanalyzedRecord (ILTR_PTRANSL tr, TIF_RECORD_VALUE_PTR pRec)
{
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;

    /*-----
    * The exdata array is where Search Keys and Linkage words live.
    * Until analyzed, the exdata is almost all zeroed.
    *-----*/
    INT32 exdata[TIF_EXDATA_PER_RECORD];
    IL_MEMSET( (IL_PANY) exdata, 0, sizeof(exdata));

    // Increment the number of records in the file
    INT32 lRecordNumber;
    lRecordNumber = TIF_lCurrentRecNum = TIF_TotalRecordCount++;

    exdata[TIF_FLAGS_SLOT] = TIF_origin | TIF_IS_UNANALYZED;

    exdata[TIF_NEXT_IN_CIG_SLOT] = lRecordNumber; // singleton CIG
    exdata[TIF_NEXT_IN_SKG_SLOT] = lRecordNumber; // singleton SKG
    exdata[TIF_NEXT_IN_FIG_SLOT] = lRecordNumber; // singleton FIG

    int rc = ILDFX_AddRecord ( TIF_hFile, lRecordNumber,
                              pRec, TIFREC_SIZE(pRec), exdata );

    return rc;
} //---- StoreUnanalyzedRecord

/*-----
* Name:      AnalyzeAndStoreRecord (called from TIFPutRecord)
*
* Purpose: Analyze and Store a record.
*           Establish membership in CIG and/or SKG as appropriate.
*
* Details:
*           This function always does analysis to set CIG & SKG membership
*           Then it effects one of the following storage options:
*           1. add new record (body + exdata)
*           2. update existing record (body + exdata)
*           3. update existing record (exdata only)
*
* The 'ItemToIgnore' recordnumber was added as an afterthought to allow
* the 'TIFPRO_STORE_SPECIAL_ZOMBIE' stuff to work, avoiding non-uniqueID err.
*-----*/
static int AnalyzeAndStoreRecord ( ILTR_PTRANSL tr,
                                  ILUT_PBUFFER pRecBuf,
                                  TIF_PUT_RECORD_OPTION nPutOption,
                                  BOOLEAN isaZombie,
                                  INT32 ItemToIgnore,
                                  BOOLEAN *pMustAddZombie )
{
    int rc;
    PSTTIF_TYPE pstTIF = &ILTIF_pstTIF;
    ILDFX_PHNDL phFile = TIF_hFile;
    INT32 lIDMatchIndex = TIF_NOTSET;
    INT32 lKfMatchIndex = TIF_NOTSET;
    INT32 exdata[TIF_EXDATA_PER_RECORD];
    INT32 CurrentItem;
    INT32 LastPossibleMatch;

    TIF_RECORD_VALUE_PTR pRec = (TIF_RECORD_VALUE_PTR) pRecBuf->pBuffer;

    /*-----
    * If we're going to ADD a new record, don't set CurrentItem yet, but set
    * Search Limit used when searching for ID or KF matches.
    */

```

```

/*-----*/
if ( nPutOption == TIFPRO_ANALYZE_AND_STORE_NEW_RECORD
    || nPutOption == TIFPRO_STORE_SPECIAL_ZOMBIE )
{
    CurrentItem = -1; // item# for new item not set yet
    LastPossibleMatch = TIF_TotalRecordCount - 1;
}

/*-----
 * But if we're going to UPDATE an existing record, set CurrentItem and
 * limit searches so that we never match the current item to another item
 * that sits further down in the index. (Otherwise that could happen
 * while analyzing the History File.)
 *-----*/
else
{
    CurrentItem = TIF_lCurrentRecNum;
    LastPossibleMatch = CurrentItem - 1;
}

if ( (nPutOption == TIFPRO_ANALYZE_AND_STORE_NEW_RECORD)
    || (nPutOption == TIFPRO_ANALYZE_AND_UPDATE_RECORD) )
{
    // Make sure that all fields that need to be filled are
    TIFFillDefaults (pstTIF, tr, pRecBuf);

    /*-----
     * Test code to force TIF to think that DUPLICATE IDs
     * are being supplied to it.
     *-----*/
    //static char szID[100] = "";
    //static long idlen = 0;
    //
    //if (ILTR_phase == ILTR_PHASE10)
    //{
    //    if (IL_STRING_IS_NULL(szID))
    //        IL_STRCPY (szID, TIF_FieldData(pRec, TIF_TargetIDFieldNum));
    //    else
    //        IL_STRCPY (TIF_FieldData(pRec, TIF_TargetIDFieldNum), szID);
    //}
    //TIFlogszszul ("id=%s, phase=%ld", szID, (UINT32) TIF_phase);
    //if (TIF_phase == TIF_PHASE_SANITIZING_SOURCE_RECORDS)
    //{
    //    if (IL_STRING_IS_NULL(szID))
    //    {
    //        IL_STRCPY (szID, TIF_FieldData(pRec, TIF_SourceIDFieldNum));
    //        idlen = TIF_FieldLength(pRec, TIF_SourceIDFieldNum);
    //    }
    //    else
    //    {
    //        IL_STRCPY (TIF_FieldData(pRec, TIF_SourceIDFieldNum), szID);
    //        TIF_FieldLength(pRec, TIF_SourceIDFieldNum) = idlen;
    //    }
    //}
    //}

    /*-----
     * The exdata array is where Search Keys and Linkage words live.
     * Search Keys are Hash Values and Start&End DTTM Values
     *-----*/
    IL_MEMSET((IL_PANY) exdata, 0, sizeof(exdata));

    /*-----
     * For "Special History File Update" we depend upon the exdata
     * values that are stored in the history file. The following rules
     * must be upheld:
     *
     *   all Hash IDs and DTTM values are set correctly
     *   there aren't any non-unique IDs.
     *   each item belongs to a singleton CIG and a singleton SKG
     *   each item has flags == FROM_PREVIOUS
     *
     * Now simply copy the exdata that has been read from the history file.
     *-----*/
    if (nPutOption == TIFPRO_SPECIAL_HISTORY_FILE_UPDATE)

```

```

{
    for (int i=0; i < TIF_EXDATA_PER_RECORD; i++)
        exdata[i] = TIFX(phFile, CurrentItem, i);

    //---- For Appts, set the OUT_OF_RANGE flag if applicable
    if (ILTR_nFunction == ILTR_APPT)
    {
        rc = CheckAndSetOutOfRange (tr, pstTIF, pRec, CurrentItem, exdata);
        if (rc != SUCCESS) return rc;
    }
}

/*-----
 * For all other flavors of 'PutRecord', we need to compute exdata
 * values, and when doing synchronization we look for ID-based matches.
 * Also set the isRecurringItem flag, and sanitize Exclusion Lists
 * for Recurring Items.
 *-----*/
else
{
    /*-----
     * Use ZOMBIE flag to mark FastSync DELETE items.  Zombie items play
     * an important but very limited "placeholder" role.  They never are
     * party to KeyField-based matches, and when a zombie is matched up
     * with a P-Item we don't set any of the P-Item's ID-match flag bits.
     * See tifsync2\TIFSyncDigestFastLoad for further use of zombies.
     *-----*/
    if (isaZombie)
        exdata[TIF_FLAGS_SLOT] = TIF_ZOMBIE;

    rc = TIFComputeSearchKeyValues(pstTIF, pRec, exdata);
    if (rc != SUCCESS)
        return rc;

    if ( (TIF_origin == TIF_FROM_SOURCE)
        && (exdata[TIF_TARGETID_HASH_SLOT] != TIFHASH_NONE) )
        return TIF_ERR_TARGETID_IN_SOURCE_REC;

    if ( (TIF_origin == TIF_FROM_TARGET)
        && (exdata[TIF_SOURCEID_HASH_SLOT] != TIFHASH_NONE) )
        return TIF_ERR_SOURCEID_IN_TARGET_REC;

    if (TIF_nSynchronize)
    {
        // SYNCHRONIZATION: look for a previously loaded record, in TIF, that
        //----- has the same ID as the incoming record.
        switch(TIF_origin)
        {
            case TIF_FROM_PREVIOUS:          //-- loading Previous Sync History Records
                                              //-- check for non-unique IDs

                rc = SearchForIDMatch ( pstTIF, pRec, exdata,
                                         TIF_SOURCEID_HASH_SLOT,
                                         CurrentItem, ItemToIgnore,
                                         LastPossibleMatch,
                                         &lIDMatchIndex );
                if (rc == SUCCESS || rc == TIF_ILLEGAL_MATCH)
                    return TIF_ERR_NON_UNIQUE_SOURCEID;
                else if (rc != TIF_NO_MATCH)
                    return rc;

                rc = SearchForIDMatch ( pstTIF, pRec, exdata,
                                         TIF_TARGETID_HASH_SLOT,
                                         CurrentItem, ItemToIgnore,
                                         LastPossibleMatch,
                                         &lIDMatchIndex );
                if (rc == SUCCESS || rc == TIF_ILLEGAL_MATCH)
                    return TIF_ERR_NON_UNIQUE_TARGETID;
                else if (rc != TIF_NO_MATCH)
                    return rc;

                break;

            case TIF_FROM_TARGET:             //-- loading from Target App.

```

```

        rc = SearchForIDMatch ( pstTIF, pRec, exdata,
                                TIF_TARGETID_HASH_SLOT,
                                CurrentItem, ItemToIgnore,
                                LastPossibleMatch,
                                &lIDMatchIndex );

        break;

    case TIF_FROM_SOURCE:          //-- loading from Source App.

        rc = SearchForIDMatch ( pstTIF, pRec, exdata,
                                TIF_SOURCEID_HASH_SLOT,
                                CurrentItem, ItemToIgnore,
                                LastPossibleMatch,
                                &lIDMatchIndex );

        break;

    default:
        return TIF_ERR_BAD_TIF_ORIGIN;

} //-- switch(TIF_origin)

//-- check the ID match results
if (rc == TIF_ILLEGAL_MATCH)
{
    lIDMatchIndex = TIF_NOTSET;

    //-- In FastSync mode we must add zombies for Illegal Matches
    if (TIF_bFastSyncLoad)
        *pMustAddZombie = TRUE;
}
else if (rc != SUCCESS && rc != TIF_NO_MATCH)
    return ILERROR (rc, rc);    // abnormal result

} // if (TIF_nSynchronize)

} // if (nPutOption == TIFPRO_SPECIAL_HISTORY_FILE_UPDATE...else...

//-- always put zombies and bystanders into singleton SKGs
if (isaZombie || (exdata[TIF_FLAGS_SLOT] & TIF_BYSTANDER))
    lKMatchIndex = CurrentItem;
else
{
    //-- Item isn't a zombie, so look for a KeyFields match
    rc = SearchForKeyFieldsMatch ( pstTIF, pRec,
                                   exdata[TIF_KEYFIELDS_HASH_SLOT],
                                   CurrentItem,
                                   LastPossibleMatch,
                                   &lKMatchIndex );

    if ((rc != SUCCESS) && (rc != TIF_NO_MATCH))
        return rc;    // abnormal result
}

/*-----
 * Initialize linkage words with results of ID and KeyField matching.
 * These linkage values are subsequently rationalized as we determine
 * whether the new record will be a singleton or a member of a group.
 *-----*/
exdata[TIF_NEXT_IN_CIG_SLOT] = lIDMatchIndex;    // sanitized later
exdata[TIF_NEXT_IN_SKG_SLOT] = lKMatchIndex;    // sanitized later

if ( nPutOption == TIFPRO_ANALYZE_AND_STORE_NEW_RECORD
    || nPutOption == TIFPRO_STORE_SPECIAL_ZOMBIE )
{
    // Increment the number of records in the file
    CurrentItem = TIF_lCurrentRecNum = TIF_TotalRecordCount++;
}

//-- If new item is party to an ID-match, set appropriate flag bits
if (lIDMatchIndex == TIF_NOTSET)
    exdata[TIF_FLAGS_SLOT] |= TIF_origin;
else
{
    if (TIF_origin == TIF_FROM_TARGET)
        exdata[TIF_FLAGS_SLOT] |= TIF_origin | TIF_ID_MATCH_PT;
}

```



```

        else
            exdata[TIF_FLAGS_SLOT] |= TIF_origin | TIF_ID_MATCH_PS;
    }

    switch (nPutOption)
    {
        case TIFPRO_ANALYZE_AND_STORE_NEW_RECORD:
        case TIFPRO_STORE_SPECIAL_ZOMBIE:

            // Write out the field info record to the next record in file
            rc = ILDFX_AddRecord ( phFile, CurrentItem,
                                   pRec, TIFREC_SIZE(pRec), exdata );
            break;

        case TIFPRO_ANALYZE_AND_UPDATE_RECORD:

            // Update data and exdata for current item
            rc = ILDFX_UpdateRecord ( phFile, CurrentItem,
                                       pRec, TIFREC_SIZE(pRec), exdata );
            break;

        case TIFPRO_SPECIAL_HISTORY_FILE_UPDATE:

            // Update exdata only, for current item
            rc = ILDFX_UpdateRecord ( phFile, CurrentItem,
                                       NULL, 0, exdata );
            break;

        default:
            rc = TIF_ERR_BAD_PUT_RECORD_OPTION;
    }

    if (rc != SUCCESS)
        return rc;

    /*-----
    * Mark P-item as party to an ID match, if an ID match was found.
    *-----*/
    if (lIDMatchIndex != TIF_NOTSET)
    {
        if (TIF_origin == TIF_FROM_TARGET)
            TIFX_FLAGS(phFile, lIDMatchIndex) |= TIF_ID_MATCH_PT;
        else
            TIFX_FLAGS(phFile, lIDMatchIndex) |= TIF_ID_MATCH_PS;
    }

    /*-----
    * Except for 'HistoryFileUpdate' we force all FIGs to be singletons
    *-----*/
    if (nPutOption != TIFPRO_SPECIAL_HISTORY_FILE_UPDATE)
        TIFX_NEXT_IN_FIG(phFile, CurrentItem) = CurrentItem;

    //---- cement the new member into a CIG
    rc = InitiateNewGroupMember ( phFile, CurrentItem,
                                   TIF_NEXT_IN_CIG_SLOT, "CIG" );

    //---- cement the new member into an SKG
    if (rc == SUCCESS)
        rc = InitiateNewGroupMember ( phFile, CurrentItem,
                                   TIF_NEXT_IN_SKG_SLOT, "SKG" );

    return rc;
} //---- AnalyzeAndStoreRecord

/*-----
* Name:      TIFAdjustRecordIfChanged
* Called by: GetResolution function, in TIFMERGE.CPP
* Purpose:   To make storage and search key adjustments for a Reconciled
*            Record, which the user may have edited.
*
*            If the user hasn't edited the Source Record, we don't
*            change a thing.
*-----*/

```

```

* User may have edited the record, so we need to re-compute
* the KeyFields Hash and NonKeyFields Hash and Start&End DTTM.
* If Key Fields have changed at all we need to re-compute
* SKG membership. If anything has changed we need to store
* new data record on disk, replacing old SOURCE record.
*
* Author: David Boothby, Copyright (c) IntelliLink Corporation, 1995
*/
int TIFAdjustRecordIfChanged(PSTTIF_TYPE pstTIF, INT32 Item)
{
    ILDFX_PHNDL phFile = TIF_hFile;
    TIF_RECORD_VALUE_PTR pRec = TIF_pCurrentRecord;

    /*-----*/
    * The exdata array is where Search Keys and Linkage words live.
    * Search Keys are Hash Values and Start&End DTTM Values
    *-----*/
    INT32 exdata[TIF_EXDATA_PER_RECORD];

    IL_MEMSET( (IL_PANY) exdata, 0, sizeof(exdata));

    int rc = TIFComputeSearchKeyValues(pstTIF, pRec, exdata);
    if (rc != SUCCESS)
        return rc;

    /*-----*/
    * Determine whether anything at all has changed. Check for changes both
    * to Key Fields and to Not-No-Reconcile Non-Key Fields.
    *-----*/
    if ( exdata[TIF_KEYFIELDS_HASH_SLOT] == TIFX_KEYFIELDS_HASH(phFile, Item)
        && exdata[TIF_NKFIELDS_HASH_SLOT] == TIFX_NKFIELDS_HASH(phFile, Item) )
    {
        /*----- compare in-memory record with on-disk record
        rc = TIFVerifyFieldsMatch (pstTIF, pRec, -1, Item, TIFW_ALL_FIELDS);
        if (rc != TIF_NO_MATCH)
            /*----- SUCCESS (no changes, nothing to do) or abnormal error
            return rc;
        */
    }

    /*-----*/
    * At this point we know that one or more fields have changed.
    * Now determine whether any key fields have changed...
    * We have to do this BEFORE updating the record on disk.
    *-----*/
    BOOLEAN bKeyFieldsHaveChanged;
    if (exdata[TIF_KEYFIELDS_HASH_SLOT] != TIFX_KEYFIELDS_HASH(phFile, Item))
        bKeyFieldsHaveChanged = TRUE;
    else
    {
        rc = TIFVerifyFieldsMatch (pstTIF, pRec, -1, Item, TIFW_KEY_FIELDS);
        if (rc == SUCCESS)
            bKeyFieldsHaveChanged = FALSE;
        else if (rc == TIF_NO_MATCH)
            bKeyFieldsHaveChanged = TRUE;
        else
            return rc; // abnormal error
    }

    if (bKeyFieldsHaveChanged)
        TIFlogsz("Source Record was edited; KeyFields Altered");
    else
        TIFlogsz("Source Record was edited but KeyFields are unchanged");

    /*-----*/
    * We no longer need the un-altered record. Store the altered
    * record, replacing the previous record on disk. Also store updated
    * exdata in the ILDFX index array. First we have to copy across the
    * few exdata values not already set by 'TIFComputeSearchKeyValues'.
    * Group membership will be adjusted later...
    *-----*/
    exdata[TIF_NEXT_IN_CIG_SLOT] = TIFX_NEXT_IN_CIG(phFile, Item);
    exdata[TIF_NEXT_IN_SKG_SLOT] = TIFX_NEXT_IN_SKG(phFile, Item);
    exdata[TIF_NEXT_IN_FIG_SLOT] = TIFX_NEXT_IN_FIG(phFile, Item);
    /*-----*/
    * The FLAGS slot may have TIF_ITEM_IS_RECURRING and/or TIF_WRONG_SST

```

```

    * bits set by 'TIFComputeSearchKeyValues', but we can safely get those
    * same bits from the item being adjusted, because those bits are never
    * affected by user edits.
    *-----*/
exdata[TIF_FLAGS_SLOT] = TIFX_FLAGS(phFile, Item);
/*-----
    * The FLAGS2 Slot may have the TIF2_DONE_TODO bit set or cleared
    * by 'TIFComputeSearchKeyValues'. The other bits are currently all
    * irrelevant to SmartMerge, but we copy them in for hygiene sake.
    *-----*/
exdata[TIF_FLAGS2_SLOT] &= TIF2_DONE_TODO;
exdata[TIF_FLAGS2_SLOT] |= (TIFX_FLAGS2(phFile, Item) & ~TIF2_DONE_TODO);

rc = ILDFX_UpdateRecord ( phFile, Item,
                          pRec, TIFREC_SIZE(pRec), exdata );
if (rc != SUCCESS)
    return rc;

/*-----
    * If Key Fields haven't changed, we're done!
    *-----*/
if (bKeyFieldsHaveChanged == FALSE)
    return SUCCESS;

/*-----
    * Key Fields have changed, so we have to change SKG membership.
    * First remove item from the SKG that it currently belongs to.
    *-----*/
rc = TIFremoveFromSKG (phFile, Item, NULL);
if (rc != SUCCESS)
    return rc;

/*-----
    * Now look for a different SKG to put it in...
    *-----*/
INT32 MatchingItem;
rc = SearchForKeyFieldsMatch ( pstTIF, pRec,
                              TIFX_KEYFIELDS_HASH(phFile, Item),
                              Item, -1, // search entire index
                              &MatchingItem );

if (rc == TIF_NO_MATCH)
    return SUCCESS; // all done; Item is now in its own singleton SKG
else if (rc != SUCCESS)
    return rc; //---- abnormal error

/*-----
    * Found a KF Match; add item to the SKG that contains the Matching Item
    *-----*/
TIFX_NEXT_IN_SKG(phFile, Item) = MatchingItem;
rc = InitiateNewGroupMember (phFile, Item, TIF_NEXT_IN_SKG_SLOT, "SKG");
return rc;
} //---- TIFAdjustRecordIfChanged

/*-----
    * Name:      VerifyIDMatch
    *
    * Compare full ID field values to verify match.
    * This is an expensive operation, done only when ID Hash values match.
    *
    * WARNING:  uses the TIF_CurrentField and TIF_OriginalField buffers
    *-----*/
static int VerifyIDMatch (PSTTIF_TYPE pstTIF,
                          TIF_RECORD_VALUE_PTR pRec,
                          INT32 SecondRecordIndex,
                          int IDSlot)
{
    /*-----
    * Use 'secondRecordIndex' to read second record from ILDFX file.
    * First record is held in memory as *pRec. Iterate thru
    * all fields, looking for key fields. For each key field compare the
    * field values found in the two records. Return SUCCESS iff
    * ID field values match exactly.
    *-----

```

```

*
* Other return values are TIF_NO_MATCH, or other error codes for
* abnormal error conditions.
*-----*/
int rc;

INT32 FirstIDLength;
INT32 SecondIDLength;
INT16 IDFieldNum;

if (IDSlot == TIF_SOURCEID_HASH_SLOT)
    IDFieldNum = TIF_SourceIDFieldNum;
else
    IDFieldNum = TIF_TargetIDFieldNum;

rc = TIFRetrieveFieldByIndex ( pstTIF,
                              IDFieldNum,
                              &FirstIDLength,
                              &TIF_CurrentField,
                              pRec );

if (rc != SUCCESS)
    return rc; //---- abnormal error

rc = TIFRetrieveRecord (pstTIF, SecondRecordIndex, &TIF_SecondRecord);
if (rc == SUCCESS)
{
    //---- we arbitrarily choose to use 'TIF_OriginalField' buffer here
    rc = TIFRetrieveFieldByIndex ( pstTIF,
                                  IDFieldNum,
                                  &SecondIDLength,
                                  &TIF_OriginalField,
                                  TIF_pSecondRecord );

    if (rc == SUCCESS)
    {
        if (TIFFieldValuesDiffer( pstTIF,
                                  (IL_PSTR) TIF_pCurrentField,    // first ID
                                  (IL_PSTR) TIF_pOriginalField,    // 2nd ID
                                  FirstIDLength, SecondIDLength,
                                  IDFieldNum ))

            rc = TIF_NO_MATCH;
        else
            rc = SUCCESS;
    }
}

return rc;
} //---- VerifyIDMatch

/*-----*/
* Function: SearchForIDMatch
*
* Search for any item other than 'ExcludedItems (1 & 2)' that has same ID
* as CurrentRecord. Can search for SourceID or TargetID match.
*
* If we find an ID match between two non-zombie items where one is Simple
* and the other is Recurring, report an ILLEGAL MATCH.
*-----*/
static int SearchForIDMatch (PSTTIF_TYPE pstTIF,
                             TIF_RECORD_VALUE_PTR pRec,
                             INT32 *exdata,           // in (array)
                             int slotIndex,            // sourceID or targetID
                             INT32 ExcludedItem1,
                             INT32 ExcludedItem2,
                             INT32 LastPossibleMatch,
                             INT32 *pMatchIndex)      // out
{
    INT32 lFlags;
    ILDFX_PHNDL phFile = TIF_hFile;
    ILDFX_EC ec;
    INT32 lKey = TIF_FIRST_ITEMNO - 1; // (so that lKey+1 is First Item#)

    INT32 lHashSought = exdata[slotIndex];
    if (lHashSought == TIFHASH_NONE)

```

```

    return TIF_NO_MATCH; // zero-length IDs don't match...otherwise we'd
                        // see lots of non-uniqueID errors.

//---- make our item exclusion logic as efficient as possible
if (ExcludedItem1 == -1)
    ExcludedItem1 = ExcludedItem2;

//-----TIF/ILDFX/IndexScanningLoop
for (;;)
{
    INT32 lNextKey = lKey + 1;
    ec = ILDFX_FindNextExDataMatch (phFile,
                                    lNextKey,
                                    ExcludedItem1,
                                    LastPossibleMatch,
                                    lHashSought,
                                    slotIndex,      // ExData[] index
                                    &lKey);          // OUT: found key

    if (ec != ILDFX_OK)
        break;

    //---- skip over 2nd excluded item
    if (lKey == ExcludedItem2)
        continue;

    lFlags = TIFX_FLAGS(phFile, lKey);

    //---- skip over any unanalyzed/garbage records
    //---- but do not skip over zombie or bystander records here
    if (lFlags & (TIF_IS_UNANALYZED | TIF_IS_GARBAGE))
        continue;

    //--- now compare full IDs for equality
    ec = VerifyIDMatch (pstTIF, pRec, lKey, slotIndex);
    if (ec != TIF_NO_MATCH)
        break;          // leave loop on SUCCESS or abnormal error
}

if (ec == ILDFX_END_OF_INDEX)
    return TIF_NO_MATCH;
else if (ec != SUCCESS)
    return ec;

/*-----
 * Found a match, now make sure it's unique.
 * When an ID match is found, we OR a bit into the FLAGS slot, for both
 * partners in the match. Here we check to make sure that the bit
 * isn't already set. If it is, we have a non-unique ID problem.
 *-----*/
INT32 lMask;
switch (slotIndex)
{
    case TIF_TARGETID_HASH_SLOT:
        lMask = TIF_ORIGIN_MASK | TIF_ID_MATCH_PT;
        break;

    case TIF_SOURCEID_HASH_SLOT:
        lMask = TIF_ORIGIN_MASK | TIF_ID_MATCH_PS;
        break;

    default:
        return TIF_ERR_BAD_ID_SLOT;
}

if ((lFlags & lMask) != TIF_FROM_PREVIOUS)
{
    if (slotIndex == TIF_SOURCEID_HASH_SLOT)
        return TIF_ERR_NON_UNIQUE_SOURCEID;
    else
        return TIF_ERR_NON_UNIQUE_TARGETID;
}

//---- If incoming item matches a BYSTANDER, kill the bystander
//---- by turning it into garbage. The new input, which may or
//---- may not be a zombie, will replace the old bystander.
if (lFlags & TIF_BYSTANDER)

```

```

    {
        TIFloglint (90, "Killing bystander #lld", lKey);
        TIFX_FLAGS(phFile, lKey) &= ~TIF_BYSTANDER;
        TIFX_FLAGS(phFile, lKey) |= TIF_IS_GARBAGE;
        return TIF_NO_MATCH;
    }

    /*-----
    * Tell caller which Previous Item the ID match points to (whether we
    * end up reporting SUCCESS or ILLEGAL MATCH).
    *-----*/
    *pMatchIndex = lKey;

    /*-----
    * If an ID match is found between two non-zombie items, where one is
    * recurring and the other is non-recurring, report an ILLEGAL match.
    * This will ensure that we never have to process UPDATES from Simple
    * to Recurring, or vice versa. Instead such an UPDATE is processed as
    * an ADD and a DELETE. This is to just to "KISS" off a host of
    * complicated UPDATE cases involving FANNING, etc.)
    *-----*/
    {
        INT32 lZombie = exdata[TIF_FLAGS_SLOT] & TIF_ZOMBIE;
        INT32 lRecur1 = exdata[TIF_FLAGS_SLOT] & TIF_ITEM_IS_RECURRING;
        INT32 lRecur2 = lFlags & TIF_ITEM_IS_RECURRING;

        if (lZombie == 0 && lRecur1 != lRecur2)
            return TIF_ILLEGAL_MATCH;

        /*---- Here we have an unimpeachable ID match
        else
            return SUCCESS;
    }

} //---- SearchForIDMatch

/*-----
* Function: SearchForKeyFieldsMatch
*
* Search for any item other than 'ExcludedItem' that has same Key Fields
* as CurrentRecord.
*-----*/
static int SearchForKeyFieldsMatch (PSTTIF_TYPE pstTIF,
                                    TIF_RECORD_VALUE_PTR pRec,
                                    INT32 lHashSought,
                                    INT32 ExcludedItem,
                                    INT32 LastPossibleMatch,
                                    INT32 *pMatchIndex)
{
    INT32 lFlags;
    ILDFX_PHNDL phFile = TIF_hFile;
    ILDFX_EC ec;
    INT32 lKey = TIF_FIRST_ITEMNO - 1; // (so that lKey+1 is First Item#)

    /*-----TIF/ILDFX/IndexScanningLoop
    for (;;)
    {
        INT32 lNextKey = lKey + 1;
        ec = ILDFX_FindNextExDataMatch (phFile,
                                        lNextKey,
                                        ExcludedItem,
                                        LastPossibleMatch,
                                        lHashSought,
                                        TIF_KEYFIELDS_HASH_SLOT, // in ExData[]
                                        &lKey); // OUT: found key

        if (ec != ILDFX_OK)
            break;

        lFlags = TIFX_FLAGS(phFile, lKey);

        /*---- skip over any zombie or unanalyzed or garbage or bystander records
        if (lFlags & TIF_IGNORE)
            continue;
    }
}

```

```

        ec = TIFVerifyFieldsMatch (pstTIF, pRec, -1, lKey, TIFW_KEY_FIELDS);
        if (ec != TIF_NO_MATCH)
            break;          // leave loop on SUCCESS or abnormal error
    }

    if (ec == SUCCESS)
    {
        *pMatchIndex = lKey;
        return SUCCESS;
    }
    else if (ec == ILDFX_END_OF_INDEX)
        return TIF_NO_MATCH;
    else
        return ec;          // abnormal error
} //---- SearchForKeyFieldsMatch

/*-----
* Name:      EffectiveFieldLength
* Purpose:   Get field length, adjusted for whether it is BINARY or not.
*           For non-zero length non-binary fields, subtract one from
*           length to exclude the null terminator.
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*-----*/
static INT32 EffectiveFieldLength ( PSTTIF_TYPE pstTIF,
                                   TIF_RECORD_VALUE_PTR pRecord,
                                   int fieldnum )
{
    INT32 lFieldLength = TIF_FieldLength(pRecord, fieldnum);
    if (lFieldLength == 0)
        return 0;
    else if (TIF_FieldType(pstTIF, fieldnum) == ILX_TYPE_BINARY)
        return lFieldLength;
    else
        return lFieldLength-1;
} //---- EffectiveFieldLength

/*-----
* Name:      InitiateNewGroupMember
* Purpose:   patch new entry into circular list (CIG or SKG)
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*
* NOTE: we want to maintain the following ordering rules within Groups:
*
* 1.  keep same-origin items together, with all P-items first, followed
*     by all T-items, followed by all S-items.
*
* 2.  Within a set of same-origin items in a group, preserve load order.
*     The order of items in the list should be the
*     same as the order in which items join the group.
*
* This function is responsible for enforcing rules 1 and 2, regardless of
* the order in which items are loaded.  We may decide to load S-items before
* loading T-items, but this function will still maintain PTS order inside
* groups.
*
* BUT NOTE that the order of items in the ILDFX index is determined by load
* order, so if S-items are loaded before T-items, then any start-to-finish
* scan of the ILDFX index will find S-items before T-items.
*
* So, when you get results of an ILDFX index scan (e.g. from the function
* 'ILDFX_FindNextExDataMatch') do not assume that you'll hit T-items before
* hitting S-items.
*-----*/
static int InitiateNewGroupMember ( ILDFX_PHNDL phFile,
                                   INT32 RecordNumber,
                                   int LinkSlot,
                                   IL_PSTR szGroupType )
{
    INT32 MatchIndex = TIFX_NEXT_IN_GROUP(phFile, RecordNumber, LinkSlot);
    if (MatchIndex == TIF_NOTSET)

```

```

{
    //----- create a singleton group; no ordering rules apply
    TIFX_NEXT_IN_GROUP(phFile, RecordNumber, LinkSlot) = RecordNumber;

    ILLOG_logszsul ( ILDFXLOG(phFile), 95, "%s: item %ld forms singleton",
                    szGroupType, RecordNumber );
    return SUCCESS;
}
else
{
    //----- adding to a group; enforce ordering rules
    INT32 Flags = TIFX_FLAGS(phFile, RecordNumber);
    INT32 InsertType = Flags & TIF_ORIGIN_MASK;
    INT32 InsertAfter;
    INT32 InsertBefore;

    //----- Note what type to insert after, and what type to
    //----- insert before, to maintain PPPTTSSS ordering.
    switch (InsertType)
    {
        case TIF_FROM_PREVIOUS: InsertAfter = TIF_FROM_SOURCE;
                                InsertBefore = TIF_FROM_TARGET;
                                break;

        case TIF_FROM_TARGET:   InsertAfter = TIF_FROM_PREVIOUS;
                                InsertBefore = TIF_FROM_SOURCE;
                                break;

        case TIF_FROM_SOURCE:   InsertAfter = TIF_FROM_TARGET;
                                InsertBefore = TIF_FROM_PREVIOUS;
                                break;

        default: return TIF_ERR_BAD_NEWMEMBER_ORIGIN; // never happens
    }

    Flags = TIFX_FLAGS(phFile, MatchIndex);
    INT32 MatchType = Flags & TIF_ORIGIN_MASK;

    /*-----
    * Determine Insertion Rule, which depends on type of item to be
    * item to be inserted, and type of the "match item" -- i.e. the
    * group member who is sponsoring the new item.
    *
    * Note that all insertion rules allow for the degenerate case where
    * the entire group is homogeneous (all same type). In that case,
    * the new item is inserted just before the match item.
    *-----*/

    enum
    {
        INSERT_AT_END_OF_CURRENT_STREAK,
        INSERT_BEFORE_CURRENT_STREAK,
        INSERT_BEFORE_ANY_ALIEN_STREAK
    }
    Rule;

    if (InsertType == MatchType)
        Rule = INSERT_AT_END_OF_CURRENT_STREAK;
    else if (InsertBefore == MatchType)
        Rule = INSERT_BEFORE_CURRENT_STREAK;
    else
        Rule = INSERT_BEFORE_ANY_ALIEN_STREAK;

    /*-----
    * Example of "insert before any alien streak":
    * suppose we're trying to insert a T-item, and the matchType is P.
    * We don't know whether the group currently contains any T-items
    * or S-items; all we know is that it contains at least one P-item.
    * So we must go forward until we hit a type boundary that marks the
    * beginning of a non-T streak. Insert the T-item just before that
    * boundary. Of course degenerate "before-match" insertion occurs
    * if group is homogeneous (all items are P-items, in this example).
    *-----*/
    //
    INT32 Current = MatchIndex;

```



```

INT32 CurrentType = MatchType;
INT32 Next;
INT32 NextType;
int i;
for (i=1; i <= TIF_MAX_GROUP_SIZE; i++) // count to guard against infinite loop
{
    Next = TIFGroup_GetNext(phFile, Current, LinkSlot);
    if (Next < 0)
        return (int) Next; // next node failed sanity check!!

    if (Next == MatchIndex) // we've come full circle...
        goto INSERT_IT_HERE;

    Flags = TIFX_FLAGS(phFile, Next);
    NextType = Flags & TIF_ORIGIN_MASK;

    //---- We always insert at a Type Boundary...
    if (NextType != CurrentType)
    {
        switch (Rule)
        {
            case INSERT_AT_END_OF_CURRENT_STREAK:
                goto INSERT_IT_HERE;

            case INSERT_BEFORE_CURRENT_STREAK:
                if (NextType == MatchType)
                    goto INSERT_IT_HERE;
                else
                    break;

            case INSERT_BEFORE_ANY_ALIEN_STREAK:
                if (NextType != InsertType)
                    goto INSERT_IT_HERE;
        }
    }

    CurrentType = NextType;
    Current = Next; // keep on circling...
}

if (i > TIF_MAX_GROUP_SIZE) // SKG list not circular ?
    return TIF_ERR_BROKEN_SKG;

INSERT_IT_HERE:
//-----

TIFX_NEXT_IN_GROUP(phFile, Current, LinkSlot) = RecordNumber;
TIFX_NEXT_IN_GROUP(phFile, RecordNumber, LinkSlot) = Next;

if (ILLOG_VERBOSE_ENOUGH (ILDFXLOG(phFile), 95))
{
    char szLog[80];
    IL_SPRINTF ( szLog, "%s: item %ld inserted between %ld and %ld",
                 szGroupType, RecordNumber, Current, Next );
    ILLOG_logsz ( ILDFXLOG(phFile), 95, szLog );
}

return SUCCESS;
}

} //---- InitiateNewGroupMember

/*-----
* Name:      TIFSetSimpleApptSpan
* Author:    David Boothby, Copyright (c) IntelliLink Corporation, 1995
*
* NOTE: several restrictions and assumptions apply. There is code in
* TIF.CPP/TIFStartNextPhase and TIFIdentifyDistinguishedFields
* which verifies that these assumptions hold.
*
* Start Date and Start Time must be absolute
* (it wouldn't be all that hard to relax that restriction,
* but for now there's no need to do so.)
*
*/

```

```

*      End Date and End Time may be absolute or relative
*
*      End Date is not required.
*
*      Midnight crossings are allowed, and are correctly handled here.
*      We allow all possible combinations of UNSPECIFIED, RELATIVE, and
*      ABSOLUTE-end dates & times. Depending on actual values, some
*      combinations may be absurd. For example, what should we do with
*      the following inputs:
*
*          Start: 10pm on 12/25/95
*          EndDate: 12/26/95 (absolute)
*          EndTime: 48 hours (relative)
*
*      The following rules are applied (there may be room for improvement...)
*
*          When a Relative EndTime and an Absolute EndDate are
*          supplied as input, ignore the EndDate input; we compute
*          EndDate ourselves in this case.
*-----*/
int TIFSetSimpleApptSpan ( PSTTIF_TYPE pstTIF,
                          TIF_RECORD_VALUE_PTR pRecord,
                          INT32 IL_DIST *plStart,
                          INT32 IL_DIST *plEnd )
{
    IL_PSTR lpszTime;
    IL_PSTR lpszDate;

    /*-----
    * Get Start Time -- assumed to be absolute
    *-----*/
    if (TIF_StartTimeFieldNum == TIF_NOTSET)
        lpszTime = "0000";
    else if (TIF_FieldOffset(pRecord, TIF_StartTimeFieldNum) == TIF_NOTSET)
        lpszTime = "0000";
    else
        lpszTime = TIF_FieldData(pRecord, TIF_StartTimeFieldNum);

    /*-----
    * Get Start Date -- assumed to be absolute
    *-----*/
    if (TIF_StartDateFieldNum == TIF_NOTSET)
        lpszDate = "19961231";
    else
        lpszDate = TIF_FieldData(pRecord, TIF_StartDateFieldNum);

    /*-----
    * combine Start Date & Time into minutes since the year 1900
    *-----*/
    *plStart = TIFConvertDateTime (lpszDate, lpszTime);

    /*-----
    * Get End Time -- may be Absolute or Relative
    *-----*/
    if (TIF_EndTimeFieldNum == TIF_NOTSET)
        lpszTime = "0000";
    else if (TIF_FieldOffset(pRecord, TIF_EndTimeFieldNum) == TIF_NOTSET)
        lpszTime = "0000";
    else
        lpszTime = TIF_FieldData(pRecord, TIF_EndTimeFieldNum);

    /*-----
    * Get End Date (Absolute or Relative) if EndDate Field is defined
    *-----*/
    INT32 DaysFromStartToEnd; DaysFromStartToEnd = TIF_NOTSET;

    if (TIF_EndDateFieldNum != TIF_NOTSET)
    {
        lpszDate = TIF_FieldData(pRecord, TIF_EndDateFieldNum);

        if (TIF_RelatedFieldNum(pstTIF, TIF_EndDateFieldNum) != TIF_NOTSET)
            /*----- EndDate is Relative...
            DaysFromStartToEnd = strtol(lpszDate, 0, 10);
        }
    }

```

```

/*-----
 * If End Time and/or Date is relative, compute absolute values
 *-----*/
if ( (TIF_EndTimeFieldNum != TIF_NOTSET)
    && (TIF_RelatedFieldNum(pstTIF, TIF_EndTimeFieldNum) != TIF_NOTSET) )
{
    //----- add duration (in minutes) to start to get end
    INT32 lMinutes = strtol(lpszTime, NULL, 10);
    *plEnd = *plStart + lMinutes;

    //---- ignore Absolute EndDate, but use Relative EndDate value (days)
    if (DaysFromStartToEnd != TIF_NOTSET)
        *plEnd += DaysFromStartToEnd * 1440L; // days * 24Hours * 60Mins/Hr
}
else
{
    //----- EndTime is Absolute; how about EndDate?
    if ( (DaysFromStartToEnd == TIF_NOTSET)
        && (TIF_EndDateFieldNum != TIF_NOTSET) )
        //----- Both EndTime and EndDate are Absolute
        *plEnd = TIFConvertDateTime(lpszDate, lpszTime);
    else
    {
        //----- EndTime is Absolute; EndDate is either Relative or UNSPECIFIED
        INT32 lEndTime = IL_AlphaToCodeTime(lpszTime) / 60;
        INT32 lStartTime = (*plStart) % 1440L;

        //----- combine StartDate with EndTime
        *plEnd = *plStart + lEndTime - lStartTime;

        if (DaysFromStartToEnd > 0)
        {
            /*-----
             * with endpoint 1 or more days after start, we don't have
             * to worry about the endTime being before the startTime.
             *-----*/
            *plEnd += (DaysFromStartToEnd * 1440L);
        }
        else if (lEndTime < lStartTime)
            //----- force a midnight crossing to avoid negative-length appt.
            *plEnd += 1440L;
    }
}

return SUCCESS;
} //---- TIFSetSimpleApptSpan

/*-----
 * Name:      SetRecurringApptSpan
 *-----*/
static int SetRecurringApptSpan ( PSTTIF_TYPE pstTIF,
                                TIF_RECORD_VALUE_PTR pRecord,
                                ILTR_PREPEAT pRepeat,
                                INT32 IL_DIST *plStart,
                                INT32 IL_DIST *plEnd )
{
    INT32      lEndDate;
    INT32      lTimeInMinutes;
    IL_PSTR    lpszTime;

    /*-----
     * Get Start Time -- assumed to be absolute
     *-----*/
    if (TIF_StartTimeFieldNum == TIF_NOTSET)
        lpszTime = "0000";
    else if (TIF_FieldOffset(pRecord, TIF_StartTimeFieldNum) == TIF_NOTSET)
        lpszTime = "0000";
    else
        lpszTime = TIF_FieldData(pRecord, TIF_StartTimeFieldNum);

    if (IL_AlphaTimeOK (lpszTime))
        lTimeInMinutes = IL_AlphaToCodeTime (lpszTime) / 60;
    else

```